HP MPI User's Guide

Sixth Edition

B6060-96004

March 2001

© Copyright 2001 Hewlett-Packard Company

Edition: Sixth

B6060-96001
Remarks: Released with HP MPI V1.7, March, 2001.

Edition: Fifth

B6060-96001
Remarks: Released with HP MPI V1.6, June, 2000.

Edition: Fourth

B6011-90001
Remarks: Released with HP MPI V1.5, February, 1999.

Edition: Third

B6011-90001
Remarks: Released with HP MPI V1.4, June, 1998.

Edition: Second

B6011-90001
Remarks: Released with HP MPI V1.3, October, 1997.

Edition: First

B6011-90001
Remarks: Released with HP MPI V1.1, January, 1997.

Notice

Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

The information contained in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Parts of this book came from Cornell Theory Center's web document. That document is copyrighted by the Cornell Theory Center.

Parts of this book came from *MPI: A Message Passing Interface*. That book is copyrighted by the University of Tennessee. These sections were copied by permission of the University of Tennessee.

Parts of this book came from *MPI Primer/Developing with LAM*. That document is copyrighted by the Ohio Supercomputer Center. These sections were copied by permission of the Ohio Supercomputer Center.

Contents

Preface ..... **xiii**
System platforms xiv
Notational conventions xv
Associated Documents xvi
Credits xvii

1 Introduction **1**
The message passing model 2
MPI concepts 3
Point-to-point communication 5
Communicators 5
Sending and receiving messages 6
Collective operations 10
Communication 10
Computation 12
Synchronization 13
MPI datatypes and packing 14
Multilevel parallelism 16
Advanced topics 16

2 Getting started **17**
Configuring your environment 18
Compiling and running your first application 19
Building and running on a single host 20
Building and running on multiple hosts 21
Running and collecting profiling data 23
Preparing mpiview instrumentation files 23
Preparing XMPI files 24
Directory structure 25

3 Understanding HP MPI **27**
Compiling applications 28
Compilation utilities 28
64-bit support 29
Thread-compliant library 30
Running applications 31
Types of applications 31
Running SPMD applications 32

Running MPMD applications 32
Runtime environment variables. 34
MPI_COMMD 35
MPI_DLIB_FLAGS 35
MPI_FLAGS. 37
MP_GANG 40
MPI_GLOBMEMSIZE. 41
MPI_INSTR 41
MPI_LOCALIP. 43
MPI_MT_FLAGS 44
MPI_NOBACKTRACE 45
MPI_REMSH 45
MPI_SHMEMCNTL 46
MPI_TMPDIR 46
MPI_WORKDIR. 46
MPI_XMPI 47
TOTALVIEW 48
Runtime utility commands. 49
`mpirun` 49
Shared library support 54
Appfiles. 55
Multipurpose daemon process. 58
Generating multihost instrumentation profiles. 59
`mpijob` 59
`mpiclean`. 61
`xmpi` 61
`mpiview`. 62
Communicating using daemons 62
IMPI. 64
Assigning hosts using LSF 64
Native Language Support. 65

4 Profiling 67

Using counter instrumentation 68
Creating an instrumentation profile 68
MPIHP_Trace_on and MPIHP_Trace_off. 69
Viewing ASCII instrumentation data 69
Viewing instrumentation data with mpiview 73
Loading an mpiview file 73
Selecting a graph type 73
Viewing multiple graphs 76
Analyzing graphs 76
Using XMPI 78
Working with postmortem mode 79

Creating a trace file. 79
Viewing a trace file 80
Working with interactive mode 90
Running an appfile 90
Changing default settings and viewing options 95
Using CXperf 100
Using the profiling interface 101
Fortran profiling interface 102

5 Tuning 103

MPI_FLAGS options 104
Message latency and bandwidth 105
Multiple network interfaces 107
Processor subscription 109
MPI routine selection 110
Multilevel parallelism 110
Coding considerations 111

6 Debugging and troubleshooting 113

Debugging HP MPI applications 114
Using a single-process debugger 114
Using a multi-process debugger 116
Limitations 117
TotalView multihost example 117
Using the diagnostics library 118
Enhanced debugging output 119
Backtrace functionality 119
Troubleshooting HP MPI applications 121
Building 122
Starting 122
Running 123
Shared memory 123
Message buffering 124
Propagation of environment variables 124
Interoperability 125
Fortran 90 programming features 125
UNIX open file descriptors 126
External input and output 126
Completing 128
Frequently asked questions 129
Time in MPI_Finalize 129
MPI clean up 129

Application hangs in MPI_Send 130

Appendix A: Example applications 131

send_receive.f 133
send_receive output 134
ping_pong.c 135
ping_pong output 137
compute_pi.f 138
compute_pi output. 139
master_worker.f90 140
master_worker output 141
cart.C. 142
cart output. 145
communicator.c. 146
communicator output 147
multi_par.f 147
io.c 156
io output. 157
thread_safe.c. 158
thread_safe output 160

Appendix B: XMPI resource file. 163

Appendix C: MPI 2.0 features supported 165

MPI I/O 166
Language interoperability 168
Thread-compliant library. 170
MPI_Init NULL arguments. 174
One-sided communication 175
Miscellaneous features. 178

Appendix D: Standard-flexibility in HP MPI 181

Glossary 185

Index 193

Figures

Figure 1 Daemon communication 63
Figure 2 ASCII instrumentation profile 71
Figure 3 MPIVIEW Graph menu. 74
Figure 4 MPIVIEW graph window 75
Figure 5 MPIVIEW Window menu 76
Figure 6 XMPI main window. 80
Figure 7 XMPI Trace Selection 81
Figure 8 XMPI trace log. 82
Figure 9 XMPI process information 85
Figure 10 XMPI Focus dialog. 86
Figure 11 XMPI Datatype dialog. 87
Figure 12 XMPI Kiviat. 89
Figure 13 XMPI Dump dialog 92
Figure 14 XMPI Express dialog. 93
Figure 15 XMPI monitor options dialog 95
Figure 16 XMPI buffer size dialog. 96
Figure 17 mpirun options dialog 97
Figure 18 Tracing options dialog. 98
Figure 19 Multiple network interfaces 108
Figure 20 Array partitioning 148

Tables

Table 1 Six commonly used MPI routines 4
Table 2 MPI blocking and nonblocking calls 9
Table 3 Organization of the /opt/mpi directory 25
Table 4 Man page categories 26
Table 5 Compilation utilities 28
Table 6 Compilation environment variables 29
Table 7 MPIVIEW analysis functions 77
Table 8 Subscription types 109
Table 9 Non-buffered messages and deadlock 124
Table 10 Example applications shipped with HP MPI 131
Table 11 MPI 2.0 features supported in HP MPI 165
Table 12 MPI I/O functionality supported by HP MPI 166
Table 13 `Info` object keys 167
Table 14 Language interoperability conversion routines 168
Table 15 HP MPI library usage 170
Table 16 Thread-initialization values 172
Table 17 Thread-support levels 172
Table 18 `Info` object routines 179
Table 19 Naming object routines 179
Table 20 HP MPI implementation of standard-flexible issues 181

Preface

This guide describes the HP MPI (version 1.7) implementation of the Message Passing Interface (MPI) standard. The guide helps you use HP MPI to develop and run parallel applications.

You should already have experience developing UNIX applications. You should also understand the basic concepts behind parallel processing, be familiar with MPI, and with the MPI 1.2 and MPI 2.0 standards (*MPI: A Message-Passing Interface Standard* and *MPI-2: Extensions to the Message-Passing Interface*, respectively).

You can access HTML versions of the MPI 1.2 and 2.0 standards at http://www.mpi-forum.org. This guide supplements the material in the MPI standards and *MPI: The Complete Reference*.

The *HP MPI User's Guide* is provided in HTML format with HP MPI. Refer to /opt/mpi/doc/html in your product. See "Directory structure" on page 25. for more information.

Some sections in this book contain command line examples used to demonstrate HP MPI concepts. These examples use the /bin/csh syntax for illustration purposes.

System platforms

HP MPI version 1.7 runs under HP-UX 11.0 or higher.
HP MPI is supported on multinode HP-UX.

The HP-UX operating system is used on:

- Workstations: s700 series
- Midrange servers: s800 series
- High-end servers

Notational conventions

This section describes notational conventions used in this book.

`bold monospace`	In command examples, **`bold monospace`** identifies input that must be typed exactly as shown.
`monospace`	In paragraph text, `monospace` identifies command names, system calls, and data structures and types. In command examples, `monospace` identifies command output, including error messages.
italic	In paragraph text, *italic* identifies titles of documents. In command syntax diagrams, *italic* identifies variables that you must provide. The following command example uses brackets to indicate that the variable *output_file* is optional: `command` *input_file* [*output_file*]
Brackets ([ ])	In command examples, square brackets designate optional entries.
KeyCap	In paragraph text, **KeyCap** indicates the keyboard keys or the user-selectable buttons on the Graphical User Interface (GUI) that you must press to execute a command.

NOTE A note highlights important supplemental information.

CAUTION **A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.**

Associated Documents

Associated documents include:

- *MPI: The Complete Reference* (2 volume set), MIT Press
- MPI 1.2 and 2.0 standards available at http://www.mpi-forum.org:
  - *MPI: A Message-Passing Interface Standard* and
  - *MPI-2: Extensions to the Message-Passing Interface*
- TotalView documents available at http://www.etnus.com:
  - *TotalView Command Line Interface Guide*
  - *TotalView User's Guide*
  - *TotalView Installation Guide*
- *CXperf User's Guide*
- *CXperf Command Reference*
- *Parallel Programming Guide for HP-UX Systems*

The following table shows World Wide Web sites that contain additional MPI information.

URL	Description
http://www.hp.com/go/mpi	Hewlett-Packard's HP MPI web page
http://www.mpi-forum.org	Official site of the MPI forum
http://www.mcs.anl.gov/Projects/mpi/index.html	Argonne National Laboratory's MPICH implementation of MPI
http://www.mpi.nd.edu/lam/	University of Notre Dame's LAM implementation of MPI
http://www.erc.msstate.edu/mpi/	Mississippi State University's MPI web page
http://www.tc.cornell.edu//Services/Edu/Topics/MPI/Basics/more.asp	Cornell Theory Center's MPI tutorial and lab exercises
http://www-unix.mcs.anl.gov/romio	Argonne National Laboratory's implementation of MPI I/O

Credits

HP MPI is based on MPICH from Argonne National Laboratory and Mississippi State University and LAM from the University of Notre Dame and Ohio Supercomputer Center.

The XMPI utility is based on LAM's version, available at http://www.mpi.nd.edu/lam/.

HP MPI includes ROMIO, a portable implementation of MPI I/O developed at the Argonne National Laboratory.

1 Introduction

This chapter provides a brief introduction about basic Message Passing Interface (MPI) concepts and the HP implementation of MPI.

This chapter contains the syntax for some MPI functions. Refer to *MPI: A Message-Passing Interface Standard* for syntax and usage details for all MPI standard functions. Also refer to *MPI: A Message-Passing Interface Standard* and to *MPI: The Complete Reference* for in-depth discussions of MPI concepts. The introductory topics covered in this chapter include:

- The message passing model
- MPI concepts
  - Point-to-point communication
  - Collective operations
  - MPI datatypes and packing
  - Multilevel parallelism
  - Advanced topics

The message passing model

Programming models are generally categorized by how memory is used. In the shared memory model each process accesses a shared address space, while in the message passing model an application runs as a collection of autonomous processes, each with its own local memory. In the message passing model processes communicate with other processes by sending and receiving messages. When data is passed in a message, the sending and receiving processes must work to transfer the data from the local memory of one to the local memory of the other.

Message passing is used widely on parallel computers with distributed memory, and on clusters of servers. The advantages of using message passing include:

- Portability—Message passing is implemented on most parallel platforms.
- Universality—Model makes minimal assumptions about underlying parallel hardware. Message-passing libraries exist on computers linked by networks and on shared and distributed memory multiprocessors.
- Simplicity—Model supports explicit control of memory references for easier debugging.

However, creating message-passing applications may require more effort than letting a parallelizing compiler produce parallel applications.

In 1994, representatives from the computer industry, government labs, and academe developed a standard specification for interfaces to a library of message-passing routines. This standard is known as MPI 1.0 (*MPI: A Message-Passing Interface Standard*). Since this initial standard, versions 1.1 (June 1995), 1.2 (July 1997), and 2.0 (July 1997) have been produced. Versions 1.1 and 1.2 correct errors and minor omissions of MPI 1.0. MPI 2.0 (*MPI-2: Extensions to the Message-Passing Interface*) adds new functionality to MPI 1.2. You can find both standards in HTML format at http://www.mpi-forum.org.

MPI-1 compliance means compliance with MPI 1.2. MPI-2 compliance means compliance with MPI 2.0. Forward compatibility is preserved in the standard. That is, a valid MPI 1.0 program is a valid MPI 1.2 program and a valid MPI 2.0 program.

MPI concepts

The primary goals of MPI are efficient communication and portability.

Although several message-passing libraries exist on different systems, MPI is popular for the following reasons:

- Support for full asynchronous communication—Process communication can overlap process computation.
- Group membership—Processes may be grouped based on context.
- Synchronization variables that protect process messaging—When sending and receiving messages, synchronization is enforced by source and destination information, message labeling, and context information.
- Portability—All implementations are based on a published standard that specifies the semantics for usage.

An MPI program consists of a set of processes and a logical communication medium connecting those processes. An MPI process cannot directly access memory in another MPI process. Inter-process communication requires calling MPI routines in both processes. MPI defines a library of routines through which MPI processes communicate.

The MPI library routines provide a set of functions that support

- Point-to-point communications
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Datatype manipulation.

Although the MPI library contains a large number of routines, you can design a large number of applications by using the six routines listed in Table 1.

Table 1 Six commonly used MPI routines

MPI routine	Description
`MPI_Init`	Initializes the MPI environment
`MPI_Finalize`	Terminates the MPI environment
`MPI_Comm_rank`	Determines the rank of the calling process within a group
`MPI_Comm_size`	Determines the size of the group
`MPI_Send`	Sends messages
`MPI_Recv`	Receives messages

You must call `MPI_Finalize` in your application to conform to the MPI Standard. HP MPI issues a warning when a process exits without calling `MPI_Finalize`.

CAUTION **There should be no code before `MPI_Init` and after `MPI_Finalize`. Applications that violate this rule are non-portable and may give incorrect results.**

As your application grows in complexity, you can introduce other routines from the library. For example, `MPI_Bcast` is an often-used routine for sending or broadcasting data from one process to other processes in a single operation. Use broadcast transfers to get better performance than with point-to-point transfers. The latter use `MPI_Send` to send data from each sending process and `MPI_Recv` to receive it at each receiving process.

The following sections briefly introduce the concepts underlying MPI library routines. For more detailed information refer to *MPI: A Message-Passing Interface Standard*.

Point-to-point communication

Point-to-point communication involves sending and receiving messages between two processes. This is the simplest form of data transfer in a message-passing model and is described in Chapter 3, “Point-to-Point Communication” in the MPI 1.0 standard.

The performance of point-to-point communication is measured in terms of total transfer time. The total transfer time is defined as

total_transfer_time = *latency* + (*message_size*/*bandwidth*)

where

latency	Specifies the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.
message_size	Specifies the size of the message in Mbytes.
bandwidth	Denotes the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in Mbytes per second.

Low latencies and high bandwidths lead to better performance.

Communicators

A communicator is an object that represents a group of processes and their communication medium or context. These processes exchange messages to transfer data. Communicators encapsulate a group of processes such that communication is restricted to processes within that group.

The default communicators provided by MPI are MPI_COMM_WORLD and MPI_COMM_SELF. MPI_COMM_WORLD contains all processes that are running when an application begins execution. Each process is the single member of its own MPI_COMM_SELF communicator.

Communicators that allow processes within a group to exchange data are termed intracommunicators. Communicators that allow processes in two different groups to exchange data are called intercommunicators.

Many MPI applications depend upon knowing the number of processes and the process rank within a given communicator. There are several communication management functions; two of the more widely used are

`MPI_Comm_size` and `MPI_Comm_rank`. The process rank is a unique number assigned to each member process from the sequence 0 through (*size*-1), where *size* is the total number of processes in the communicator.

To determine the number of processes in a communicator, use the following syntax:

```
MPI_Comm_size (MPI_Comm comm, int *size);
```

where

comm	Represents the communicator handle
size	Represents the number of processes in the group of *comm*

To determine the rank of each process in comm, use

```
MPI_Comm_rank(MPI_Comm comm, int *rank);
```

where

comm	Represents the communicator handle
rank	Represents an integer between zero and (*size* - 1)

A communicator is an argument to all communication routines. The C code example, “communicator.c” on page 146 displays the use `MPI_Comm_dup`, one of the communicator constructor functions, and `MPI_Comm_free`, the function that marks a communication object for deallocation.

Sending and receiving messages

There are two methods for sending and receiving data: blocking and nonblocking.

In blocking communications, the sending process does not return until the send buffer is available for reuse.

In nonblocking communications, the sending process returns immediately, and may only have started the message transfer operation, not necessarily completed it. The application may not safely reuse the message buffer after a nonblocking routine returns.

In nonblocking communications, the following sequence of events occurs:

1 The sending routine begins the message transfer and returns immediately.

2 The application does some computation.

3 The application calls a completion routine (for example, `MPI_Test` or `MPI_Wait`) to test or wait for completion of the send operation.

Blocking communication

Blocking communication consists of four send modes and one receive mode.

The four send modes are:

Standard (`MPI_Send`)	The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.
Buffered (`MPI_Bsend`)	The sending process returns when the message is buffered in an application-supplied buffer. Avoid using the MPI_Bsend mode because it forces an additional copy operation.
Synchronous (`MPI_Ssend`)	The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
Ready (`MPI_Rsend`)	The message is sent as soon as possible.

You can invoke any mode by using the appropriate routine name and passing the argument list. Arguments are the same for all modes.

For example, to code a standard blocking send, use

`MPI_Send (void *`*`buf`*`, int `*`count`*`, MPI_Datatype `*`dtype`*`, int `*`dest`*`, int `*`tag`*`, MPI_Comm `*`comm`*`);`

where

buf	Specifies the starting address of the buffer.
count	Indicates the number of buffer elements.

dtype	Denotes the datatype of the buffer elements.
dest	Specifies the rank of the destination process in the group associated with the communicator *comm*.
tag	Denotes the message label.
comm	Designates the communication context that identifies a group of processes.

To code a blocking receive, use

`MPI_Recv (void *`*buf*`, int` *count*`, MPI_datatype` *dtype*`, int` *source*`, int` *tag*`, MPI_Comm` *comm*`, MPI_Status *`*status*`);`

where

buf	Specifies the starting address of the buffer.
count	Indicates the number of buffer elements.
dtype	Denotes the datatype of the buffer elements.
source	Specifies the rank of the source process in the group associated with the communicator *comm*.
tag	Denotes the message label.
comm	Designates the communication context that identifies a group of processes.
status	Returns information about the received message. Status information is useful when wildcards are used or the received message is smaller than expected. Status may also contain error codes.

Examples “send_receive.f” on page 133, “ping_pong.c” on page 135, and “master_worker.f90” on page 140 all illustrate the use of standard blocking sends and receives.

NOTE

You should not assume message buffering between processes because the MPI standard does not mandate a buffering strategy. HP MPI does sometimes use buffering for `MPI_Send` and `MPI_Rsend`, but it is dependent on message size. Deadlock situations can occur when your code uses standard send operations and assumes buffering behavior for standard communication mode. Refer to “Frequently asked questions” on page 129 for an example of how to resolve a deadlock situation.

Nonblocking communication

MPI provides nonblocking counterparts for each of the four blocking send routines and for the receive routine. Table 2 lists blocking and nonblocking routine calls.

Table 2 MPI blocking and nonblocking calls

Blocking mode	Nonblocking mode
MPI_Send	MPI_Isend
MPI_Bsend	MPI_Ibsend
MPI_Ssend	MPI_Issend
MPI_Rsend	MPI_Irsend
MPI_Recv	MPI_Irecv

Nonblocking calls have the same arguments, with the same meaning as their blocking counterparts, plus an additional argument for a request.

To code a standard nonblocking send, use

`MPI_Isend(void *`*`buf`*`, int `*`count`*`, MPI_datatype `*`dtype`*`, int `*`dest`*`, int `*`tag`*`, MPI_Com `*`comm`*`, MPI_Request *`*`req`*`);`

where

req Specifies the request used by a completion routine when called by the application to complete the send operation.

To complete nonblocking sends and receives, you can use `MPI_Wait` or `MPI_Test`. The completion of a send indicates that the sending process is free to access the send buffer. The completion of a receive indicates that the receive buffer contains the message, the receiving process is free to access it, and the status object, that returns information about the received message, is set.

Collective operations

Applications may require coordinated operations among multiple processes. For example, all processes need to cooperate to sum sets of numbers distributed among them.

MPI provides a set of collective operations to coordinate operations among processes. These operations are implemented such that all processes call the same operation with the same arguments. Thus, when sending and receiving messages, one collective operation can replace multiple sends and receives, resulting in lower overhead and higher performance.

Collective operations consist of routines for communication, computation, and synchronization. These routines all specify a communicator argument that defines the group of participating processes and the context of the operation.

Collective operations are valid only for intracommunicators. Intercommunicators are not allowed as arguments.

Communication

Collective communication involves the exchange of data among all processes in a group. The communication can be one-to-many, many-to-one, or many-to-many.

The single originating process in the one-to-many routines or the single receiving process in the many-to-one routines is called the root.

Collective communications have three basic patterns:

Broadcast and Scatter	Root sends data to all processes, including itself.
Gather	Root receives data from all processes, including itself.
Allgather and Alltoall	Each process communicates with each process, including itself.

The syntax of the MPI collective functions is designed to be consistent with point-to-point communications, but collective functions are more restrictive than point-to-point functions. Some of the important restrictions to keep in mind are:

- The amount of data sent must exactly match the amount of data specified by the receiver.
- Collective functions come in blocking versions only.
- Collective functions do not use a tag argument meaning that collective calls are matched strictly according to the order of execution.
- Collective functions come in standard mode only.

For detailed discussions of collective communications refer to Chapter 4, “Collective Communication” in the MPI 1.0 standard. The following examples demonstrate the syntax to code two collective operations; a broadcast and a scatter:

To code a broadcast, use

```
MPI_Bcast(void *buf, int count, MPI_Datatype dtype, int
root, MPI_Comm comm);
```

where

buf	Specifies the starting address of the buffer.
count	Indicates the number of buffer entries.
dtype	Denotes the datatype of the buffer entries.
root	Specifies the rank of the root.
comm	Designates the communication context that identifies a group of processes.

For example “compute_pi.f” on page 138 uses MPI_BCAST to broadcast one integer from process 0 to every process in MPI_COMM_WORLD.

To code a scatter, use

`MPI_Scatter (void*` *sendbuf*`, int` *sendcount*`, MPI_Datatype` *sendtype*`, void*` *recvbuf*`, int` *recvcount*`, MPI_Datatype` *recvtype*`, int` *root*`, MPI_Comm` *comm*`);`

where

sendbuf	Specifies the starting address of the send buffer.
sendcount	Specifies the number of elements sent to each process.
sendtype	Denotes the datatype of the send buffer.
recvbuf	Specifies the address of the receive buffer.
recvcount	Indicates the number of elements in the receive buffer.
recvtype	Indicates the datatype of the receive buffer elements.
root	Denotes the rank of the sending process.
comm	Designates the communication context that identifies a group of processes.

Computation

Computational operations do global reduction operations, such as sum, max, min, product, or user-defined functions across all members of a group. There are a number of global reduction functions:

Reduce	Returns the result of a reduction at one node.
All–reduce	Returns the result of a reduction at all nodes.
Reduce-Scatter	Combines the functionality of reduce and scatter operations.
Scan	Performs a prefix reduction on data distributed across a group.

Section 4.9, “Global Reduction Operations” in the MPI 1.0 standard describes each of these functions in detail.

Reduction operations are binary and are only valid on numeric data. Reductions are always associative but may or may not be commutative.

You can select a reduction operation from a predefined list (refer to section 4.9.2 in the MPI 1.0 standard) or define your own operation. The operations are invoked by placing the operation name, for example `MPI_SUM` or `MPI_PROD`, in *op* as described in the `MPI_Reduce` syntax below.

To implement a reduction, use

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm);
```

where

sendbuf	Specifies the address of the send buffer.
recvbuf	Denotes the address of the receive buffer.
count	Indicates the number of elements in the send buffer.
dtype	Specifies the datatype of the send and receive buffers.
op	Specifies the reduction operation.
root	Indicates the rank of the root process.
comm	Designates the communication context that identifies a group of processes.

For example “compute_pi.f” on page 138 uses MPI_REDUCE to sum the elements provided in the input buffer of each process in MPI_COMM_WORLD, using MPI_SUM, and returns the summed value in the output buffer of the root process (in this case, process 0).

Synchronization

Collective routines return as soon as their participation in a communication is complete. However, the return of the calling process does not guarantee that the receiving processes have completed or even started the operation.

To synchronize the execution of processes, call `MPI_Barrier`. `MPI_Barrier` blocks the calling process until all processes in the communicator have called it. This is a useful approach for separating two stages of a computation so messages from each stage do not overlap.

To implement a barrier, use

`MPI_Barrier(MPI_Comm` *comm*);

where

comm Identifies a group of processes and a communication context.

For example, “cart.C” on page 142 uses MPI_Barrier to synchronize data before printing.

MPI datatypes and packing

You can use predefined datatypes (for example, MPI_INT in C) to transfer data between two processes using point-to-point communication. This transfer is based on the assumption that the data transferred is stored in contiguous memory (for example, sending an array in a C or Fortran application).

When you want to transfer data that is not homogeneous, such as a structure, or that is not contiguous in memory, such as an array section, you can use derived datatypes or packing and unpacking functions:

Derived datatypes
: Specifies a sequence of basic datatypes and integer displacements describing the data layout in memory. You can use user-defined datatypes or predefined datatypes in MPI communication functions.

Packing and Unpacking functions
: Provide `MPI_Pack` and `MPI_Unpack` functions so that a sending process can pack noncontiguous data into a contiguous buffer and a receiving process can unpack data received in a contiguous buffer and store it in noncontiguous locations.

Using derived datatypes is more efficient than using `MPI_Pack` and `MPI_Unpack`. However, derived datatypes cannot handle the case where the data layout varies and is unknown by the receiver, for example, messages that embed their own layout description.

Section 3.12, “Derived Datatypes” in the MPI 1.0 standard describes the construction and use of derived datatypes. The following is a summary of the types of constructor functions available in MPI:

- Contiguous (MPI_Type_contiguous)—Allows replication of a datatype into contiguous locations.
- Vector (MPI_Type_vector)—Allows replication of a datatype into locations that consist of equally spaced blocks.
- Indexed (MPI_Type_indexed)—Allows replication of a datatype into a sequence of blocks where each block can contain a different number of copies and have a different displacement.
- Structure (MPI_Type_struct)—Allows replication of a datatype into a sequence of blocks such that each block consists of replications of different datatypes, copies, and displacements.

After you create a derived datatype, you must commit it by calling `MPI_Type_commit`.

HP MPI optimizes collection and communication of derived datatypes.

Section 3.13, “Pack and unpack” in the MPI 1.0 standard describes the details of the pack and unpack functions for MPI. Used together, these routines allow you to transfer heterogeneous data in a single message, thus amortizing the fixed overhead of sending and receiving a message over the transmittal of many elements.

Refer to Chapter 3, “User-Defined Datatypes and Packing” in *MPI: The Complete Reference* for a discussion of this topic and examples of construction of derived datatypes from the basic datatypes using the MPI constructor functions.

Multilevel parallelism

By default, processes in an MPI application can only do one task at a time. Such processes are single-threaded processes. This means that each process has an address space together with a single program counter, a set of registers, and a stack.

A process with multiple threads has one address space, but each process thread has its own counter, registers, and stack.

Multilevel parallelism refers to MPI processes that have multiple threads. Processes become multithreaded through calls to multithreaded libraries, parallel directives and pragmas, and auto-compiler parallelism.

Multilevel parallelism is beneficial for problems you can decompose into logical parts for parallel execution, for example, a looping construct that spawns multiple threads to do a computation and joins after the computation is complete.

The example program, “multi_par.f” on page 147 is an example of multilevel parallelism.

Advanced topics

This chapter only provides a brief introduction to basic MPI concepts. Advanced MPI topics include:

- Error handling
- Process topologies
- User-defined datatypes
- Process grouping
- Communicator attribute caching
- The MPI profiling interface

To learn more about the basic concepts discussed in this chapter and advanced MPI topics refer to *MPI: The Complete Reference* and *MPI: A Message-Passing Interface Standard*.

2 Getting started

This chapter describes how to get started quickly using HP MPI. The semantics of building and running a simple MPI program are described, for single– and multiple–hosts. You learn how to configure your environment before running your program. You become familiar with the file structure in your HP MPI directory.

The goal of this chapter is to demonstrate the basics to getting started using HP MPI.

For complete details about running HP MPI and analyzing and interpreting profiling data, refer to Chapter 3, “Understanding HP MPI” and Chapter 4, “Profiling”. The topics covered in this chapter are:

- Configuring your environment
- Compiling and running your first application
  - Building and running on a single host
  - Building and running on multiple hosts
  - Running and collecting profiling data
- Directory structure

Configuring your environment

If you move the HP MPI installation directory from its default location in /opt/mpi:

- Set the MPI_ROOT environment variable to point to the new location.
- Set PATH to $MPI_ROOT/bin.
- Set MANPATH to $MPI_ROOT/share/man.

MPI must be installed in the same directory on every execution host.

NOTE If you have HP MPI installed on your system and want to determine its version, use the `what` command.

The `what` command returns

- The path where HP MPI is installed
- The HP MPI version number
- The date this version was released
- The product number
- The operating system version

For example:

```
% what /opt/mpi/bin/mpicc
/opt/mpi/bin/mpicc:
HP MPI 01.07.00.00 (dd/mm/yyyy) B6060BA - HP-UX 11.0
```

Compiling and running your first application

To quickly become familiar with compiling and running HP MPI programs, start with the C version of a familiar hello_world program. This program is called hello_world.c and prints out the text string "Hello world! I'm *r* of *s* on *host*" where *r* is a process's rank, *s* is the size of the communicator, and *host* is the host on which the program is run. The processor name is the host name for this implementation.

The source code for hello_world.c is stored in /opt/mpi/help and is shown below.

```
#include <stdio.h>
#include <mpi.h>

void main(argc, argv)

int                     argc;
char                    *argv[];

{

        int             rank, size, len;

        char            name[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Comm_size(MPI_COMM_WORLD, &size);

        MPI_Get_processor_name(name, &len);

        printf("Hello world!I'm %d of %d on %s\n", rank, size,
                                            name);

        MPI_Finalize();

        exit(0);

}
```

Building and running on a single host

This example teaches you the basic compilation and run steps to execute hello_world.c on your local host with four-way parallelism. To build and run hello_world.c on a local host named jawbone:

Step 1. Change to a writable directory.

Step 2. Compile the hello_world executable file:

```
% mpicc -o hello_world /opt/mpi/help/hello_world.c
```

Step 3. Run the hello_world executable file:

```
% mpirun -np 4 hello_world
```

where `-np 4` specifies the number of processes to run is 4.

Step 4. Analyze hello_world output.

HP MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

```
Hello world! I'm 1 of 4 on jawbone
Hello world! I'm 3 of 4 on jawbone
Hello world! I'm 0 of 4 on jawbone
Hello world! I'm 2 of 4 on jawbone
```

Building and running on multiple hosts

This example teaches you to build and run hello_world.c using two hosts to achieve four-way parallelism. For this example, the local host is named jawbone and a remote host is named wizard. This assumes that both machines run either HP-UX or hello_world.c is built on HP-UX so the same binary file can run on both hosts. To build and run hello_world.c on two hosts, use the following procedure, replacing jawbone and wizard with the names of your machines:

Step 1. Edit the .rhosts file on jawbone and wizard.

Add an entry for wizard in the .rhosts file on jawbone and an entry for jawbone in the .rhosts file on wizard. In addition to the entries in the .rhosts file, ensure that your remote machine permissions are set up so that you can use the `remsh` command to that machine. Refer to the HP-UX remsh(1) man page for details.

You can use the `MPI_REMSH` environment variable to specify a command other than `remsh` to start your remote processes. Refer to "MPI_REMSH" on page 45. Ensure that the correct commands and permissions are set up on all hosts.

Step 2. Change to a writable directory.

Step 3. Compile the hello_world executable:

```
% mpicc -o hello_world /opt/mpi/help/hello_world.c
```

Step 4. Copy the hello_world executable file from jawbone to a directory on wizard that is in your command path ($PATH).

Step 5. Create an appfile.

An appfile is a text file that contains process counts and a list of programs. In this example, create an appfile named my_appfile containing the following two lines:

```
-np 2 hello_world
-h wizard -np 2 hello_world
```

The appfile file should contain a separate line for each host. Each line specifies the name of the executable file and the number of processes to run on the host. The `-h` option is followed by the name of the host where the specified processes must be run. Instead of using the host name, you may use its IP address.

Step 6. Run the hello_world executable file:

```
% mpirun -f my_appfile
```

The `-f` option specifies the filename that follows it is an appfile. `mpirun` parses the appfile, line by line, for the information to run the program. In this example, mpirun runs the hello_world program with two processes on the local machine, jawbone, and two processes on the remote machine, wizard, as dictated by the `-np 2` option on each line of the appfile.

Step 7. Analyze hello_world output.

HP MPI prints the output from running the hello_world executable in non-deterministic order. The following is an example of the output:

```
Hello world! I'm 2 of 4 on wizard
Hello world! I'm 0 of 4 on jawbone
Hello world! I'm 3 of 4 on wizard
Hello world! I'm 1 of 4 on jawbone
```

Notice that processes 0 and 1 run on jawbone, the local host, while processes 2 and 3 run on wizard. HP MPI guarantees that the ranks of the processes in MPI_COMM_WORLD are assigned and sequentially ordered according to the order the programs appear in the appfile. The appfile in this example, my_appfile, describes the local host on the first line and the remote host on the second line.

Running and collecting profiling data

When you run your hello_world program, as described in “Compiling and running your first application” on page 19, you can set options so that you collect counter instrumentation and profiling data to view and analyze using the mpiview and XMPI utilities.

This section describes the mpirun options you can use to collect instrumentation data. For complete details about how to use the mpiview and XMPI utilities to analyze profiling information refer to Chapter 4, “Profiling”.

Preparing mpiview instrumentation files

Counter instrumentation provides cumulative statistics about your applications. Once you have created an instrumentation profile, you can view the data either in ASCII format or graphically using the mpiview utility.

To create instrumentation files in both formats when you run the hello_world program enter:

```
% mpirun -i hello_world -np 4 hello_world
```

where

`-i hello_world`	Enables runtime instrumentation profiling for all processes and uses the name following the `-i` option (in this case, `hello_world`) as the prefix to your instrumentation file.
`-np 4`	Specifies the number of processes.
`hello_world`	Specifies the name of the executable.

This invocation creates an instrumentation profile in two formats, each with the prefix hello_world as defined by the `-i` option: hello_world.instr is in ASCII format, and hello_world.mpiview is in graphical format. You can use the mpiview utility to analyze the .mpiview format.

Preparing XMPI files

You can use XMPI in either interactive or postmortem mode. To use XMPI's postmortem mode, you must first create a trace file. Load this file into XMPI to view state information for each process in your application. The following example shows you how to create the trace file, but for details about using XMPI in postmortem and interactive mode, refer to "Using XMPI" on page 78.

When you run your hello_world program and want to create instrumentation files to use with the XMPI utility enter:

```
% mpirun -t hello_world -np 4 hello_world
```

where

`-t hello_world`	Enables run time raw trace generation for all processes and uses the name following the `-t` option (in this case, `hello_world`) as the prefix to your instrumentation file.
`-np` *#*	Specifies the number of processes to run.
`hello_world`	Specifies the name of the executable to run.

`mpirun` creates a raw trace dump for each application process and uses the name following the `-t` option, in this case, `hello_world,` as the prefix for each file. `MPI_Finalize` consolidates all the raw trace dump files into a single file, hello_world.tr. Load hello_world.tr into XMPI for analysis.

Directory structure

All HP MPI files are stored in the /opt/mpi directory. The directory structure is organized as described in Table 3.

If you move the HP MPI installation directory from its default location in /opt/mpi, set the `MPI_ROOT` environment variable to point to the new location. Refer to "Configuring your environment" on page 18.

Table 3 Organization of the /opt/mpi directory

Subdirectory	Contents
bin	Command files for the HP MPI utilities
doc/html	The *HP MPI User's Guide*
help	Source files for the example programs
include	Header files
lib/X11/app-defaults	Application default settings for the XMPI trace utility and the mpiview profiling tool
lib/pa1.1	MPI PA-RISC 32-bit libraries
lib/pa20_64	MPI PA-RISC 64-bit libraries
lib/hpux32	MPI Itanium 32-bit libraries
lib/hpux64	MPI Itanium 64-bit libraries
newconfig/	Configuration files and release notes
share/man/man1.Z	Man pages for the HP MPI utilities
share/man/man3.Z	Man pages for HP MPI library

The man pages located in the /opt/mpi/share/man/man1.Z subdirectory can be grouped into three categories: general, compilation, and run time. There is one general man page, MPI.1, that is an overview describing general features of HP MPI. The compilation and run-time man pages are those that describe HP MPI utilities.

Table 4 describes the three categories of man pages in the man1.Z subdirectory that comprise man pages for HP MPI utilities.

Table 4 Man page categories

Category	man pages	Description
General	MPI.1	Describes the general features of HP MPI
Compilation	mpicc.1, mpiCC.1, mpif77.1, mpif90.1	Describes the available compilation utilities. Refer to “Compiling applications” on page 28 for more information
Runtime	mpiclean.1, mpijob.1, mpirun.1,mpiview.1, xmpi.1, mpienv.1, mpidebug.1, mpimtsafe.1	Describes runtime utilities, environment variables, debugging, thread-safe and diagnostic libraries.

3 Understanding HP MPI

This chapter provides information about the HP MPI implementation of MPI. The topics covered include details about compiling and running your HP MPI applications:

- Compiling applications
  - Compilation utilities
  - 64-bit support
  - Thread-compliant library
- Running applications
  - Types of applications
  - Runtime environment variables
  - Runtime utility commands
  - Communicating using daemons
  - IMPI
  - Assigning hosts using LSF
  - Native Language Support

Compiling applications

The compiler you use to build HP MPI applications depends upon which programming language you use. The HP MPI compiler utilities are shell scripts that invoke the appropriate native compiler. You can pass the pathname of the MPI header files using the `-I` option and link an MPI library (for example, the diagnostic or thread-compliant library) using the `-Wl`, `-L` or `-l` option.

By default, HP MPI compiler utilities include a small amount of debug information in order to allow the TotalView debugger to function. However, certain compiler options are incompatible with this debug information. Use the `-notv` option to exclude debug information. The `-notv` option will also disable TotalView usage on the resulting executable. The `-notv` option applies to archive libraries only.

Compilation utilities

HP MPI provides separate compilation utilities and default compilers for the languages shown in Table 5.

Table 5 Compilation utilities

Language	Utility	Default compiler
C	`mpicc`	/opt/ansic/bin/cc
C++	`mpiCC`	/opt/aCC/bin/aCC
Fortran 77	`mpif77`	/opt/fortran/bin/f77
Fortran 90	`mpif90`	/opt/fortran90/bin/f90

If aCC is not available, `mpiCC` uses CC as the default C++ compiler.

Even though the `mpiCC` and `mpif90` compilation utilities are shipped with HP MPI, all C++ and Fortran 90 applications use C and Fortran 77 bindings respectively.

If you want to use a compiler other than the default one assigned to each utility, set the corresponding environment variables shown in Table 6.

Table 6 **Compilation environment variables**

Utility	Environment variable
mpicc	MPI_CC
mpiCC	MPI_CXX
mpif77	MPI_F77
mpif90	MPI_F90

CAUTION **HP MPI does not support applications that are compiled with the following options:**

- +autodblpad—Fortran 77 programs
- +autodbl—Fortran 90 programs
- +autodbl4—Fortran 90 programs

64-bit support

HP-UX 11.0 is available as a 32- and 64-bit operating system. You must run 64-bit executables on the 64-bit system (though you can build 64-bit executables on the 32-bit system).

HP MPI supports a 64-bit version of the MPI library on platforms running HP-UX 11.0. Both 32- and 64-bit versions of the library are shipped with HP-UX 11.0. For HP-UX 11.0, you cannot mix 32-bit and 64-bit executables in the same application.

The mpicc and mpiCC compilation commands link the 64-bit version of the library if you compile with the +DA2.0W or +DD64 options. Use the following syntax:

```
[mpicc | mpiCC] [+DA2.0W | +DD64] -o filename filename.c
```

When you use mpif90, compile with the +DA2.0W option to link the 64-bit version of the library. Otherwise, mpif90 links the 32-bit version. For example, to compile the program myprog.f90 and link the 64-bit library enter:

```
% mpif90 +DA2.0W -o myprog myprog.f90
```

Thread-compliant library

HP MPI provides a thread-compliant library for applications running under HP-UX 11.0 (32- and 64-bits). By default, the non thread-compliant library (libmpi) is used when running HP MPI jobs. Linking to the thread-compliant library (libmtmpi) is now required only for applications that have multiple threads making MPI calls simultaneously. In previous releases, linking to the thread-compliant library was required for multithreaded applications even if only one thread was making a MPI call at a time. See Table 15 on page 170.

Application types that no longer require linking to the thread-compliant library include:

- +O3 +Oparallel
- Thread parallel MLIB applications
- OpenMP
- pthreads (Only if no two threads call MPI at the same time. Otherwise, use the thread-compliant library for pthreads.)

Running applications

This section introduces the methods to run your HP MPI application. Using one of the mpirun methods is required. The examples below demonstrate two basic methods. Refer to “mpirun” on page 49 for all the mpirun command line options.

You should use the `-j` option to display the HP MPI job ID. The job ID is useful during troubleshooting to check for a hung job using `mpijob` or terminate a job using `mpiclean`.

There are two methods you can use to start your application:

- Use `mpirun` with the `-np #` option and the name of your program. For example,

  ```
  % mpirun -j -np 4 hello_world
  ```

  starts an executable file named hello_world with four processes. This is the recommended method to run applications on a single host with a single executable file.

- Use `mpirun` with an appfile. For example,

  ```
  % mpirun -f appfile
  ```

  where `-f appfile` specifies a text file (appfile) that is parsed by `mpirun` and contains process counts and a list of programs.

  You can use an appfile when you run a single executable file on a single host and you must use this appfile method when you run on multiple hosts or run multiple executables. For details about building your appfile, refer to “Creating an appfile” on page 55.

NOTE Starting an application without using the `mpirun` command is no longer supported.

Types of applications

HP MPI supports two programming styles: SPMD applications and MPMD applications.

Running SPMD applications

A single program multiple data (SPMD) application consists of a single program that is executed by each process in the application. Each process normally acts upon different data. Even though this style simplifies the execution of an application, using SPMD can also make the executable larger and more complicated.

Each process calls `MPI_Comm_rank` to distinguish itself from all other processes in the application. It then determines what processing to do.

To run a SPMD application, use the `mpirun` command like this:

```
% mpirun -np # program
```

where # is the number of processors and *program* is the name of your application.

Suppose you want to build a C application called poisson and run it using five processes to do the computation. To do this, use the following command sequence:

```
% mpicc -o poisson poisson.c
% mpirun -np 5 poisson
```

Running MPMD applications

A multiple program multiple data (MPMD) application uses two or more separate programs to functionally decompose a problem.

This style can be used to simplify the application source and reduce the size of spawned processes. Each process can execute a different program.

To run an MPMD application, the `mpirun` command must reference an appfile that contains the list of programs to be run and the number of processes to be created for each program.

A simple invocation of an MPMD application looks like this:

```
% mpirun -f appfile
```

where *appfile* is the text file parsed by `mpirun` and contains a list of programs and process counts.

Suppose you decompose the poisson application into two source files: poisson_master (uses a single master process) and poisson_child (uses four child processes).

The appfile for the example application contains the two lines shown below (refer to "Creating an appfile" on page 55 for details).

```
-np 1 poisson_master
-np 4 poisson_child
```

To build and run the example application, use the following command sequence:

```
% mpicc -o poisson_master poisson_master.c
% mpicc -o poisson_child poisson_child.c
% mpirun -f appfile
```

See "Creating an appfile" on page 55 for more information about using appfiles.

Runtime environment variables

Environment variables are used to alter the way HP MPI executes an application. The variable settings determine how an application behaves and how an application allocates internal resources at runtime.

Many applications run without setting any environment variables. However, applications that use a large number of nonblocking messaging requests, require debugging support, or need to control process placement may need a more customized configuration.

Environment variables are always local to the system where `mpirun` runs. To propagate environment variables to remote hosts, specify each variable in an appfile using the `-e` option. See “Creating an appfile” on page 55 for more information.

The environment variables that affect the behavior of HP MPI at runtime are listed below and described in the following sections:

- `MPI_COMMD`
- `MPI_DLIB_FLAGS`
- `MPI_FLAGS`
- `MP_GANG`
- `MPI_GLOBMEMSIZE`
- `MPI_INSTR`
- `MPI_LOCALIP`
- `MPI_MT_FLAGS`
- `MPI_NOBACKTRACE`
- `MPI_REMSH`
- `MPI_SHMEMCNTL`
- `MPI_TMPDIR`
- `MPI_WORKDIR`
- `MPI_XMPI`
- `TOTALVIEW`

MPI_COMMD

`MPI_COMMD` routes all off-host communication through daemons rather than between processes. The `MPI_COMMD` syntax is as follows:

out_frags`,`*in_frags*

where

out_frags Specifies the number of 16Kbyte fragments available in shared memory for outbound messages. Outbound messages are sent from processes on a given host to processes on other hosts using the communication daemon.

The default value for *out_frags* is 64. Increasing the number of fragments for applications with a large number of processes improves system throughput.

in_frags Specifies the number of 16Kbyte fragments available in shared memory for inbound messages. Inbound messages are sent from processes on one or more hosts to processes on a given host using the communication daemon.

The default value for *in_frags* is 64. Increasing the number of fragments for applications with a large number of processes improves system throughput.

Refer to “Communicating using daemons” on page 62 for more information.

MPI_DLIB_FLAGS

`MPI_DLIB_FLAGS` controls runtime options when you use the diagnostics library. The `MPI_DLIB_FLAGS` syntax is a comma separated list as follows:

`[ns,][h,][strict,][nmsg,][nwarn,][dump:`*prefix*`,]`
`[dumpf:`*prefix*`][x`*NUM*`]`

where

`ns` Disables message signature analysis.

`h` Disables default behavior in the diagnostic library that ignores user specified error handlers. The default considers all errors to be fatal.

`strict`	Enables MPI object-space corruption detection. Setting this option for applications that make calls to routines in the MPI 2.0 standard may produce false error messages.
`nmsg`	Disables detection of multiple buffer writes during receive operations and detection of send buffer corruptions.
`nwarn`	Disables the warning messages that the diagnostic library generates by default when it identifies a receive that expected more bytes than were sent.
`dump:`*prefix*	Dumps (unformatted) all sent and received messages to *prefix*.msgs.*rank* where rank is the rank of a specific process.
`dumpf:`*prefix*	Dumps (formatted) all sent and received messages to *prefix*.msgs.*rank* where rank is the rank of a specific process.
`x`*NUM*	Defines a type-signature packing size. *NUM* is an unsigned integer that specifies the number of signature leaf elements. For programs with diverse derived datatypes the default value may be too small. If *NUM* is too small, the diagnostic library issues a warning during the MPI_Finalize operation.

Refer to “Using the diagnostics library” on page 118 for more information.

MPI_FLAGS

MPI_FLAGS modifies the general behavior of HP MPI. The MPI_FLAGS syntax is a comma separated list as follows:

```
[edde,][exdb,][egdb,][eadb,][ewdb,][l,][f,]
[s[a|p][#],][y[#],][o,][+E2,][C,][D,][E,][z]
```

where

edde	Starts the application under the dde debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 114 for more information.
exdb	Starts the application under the xdb debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 114 for more information.
egdb	Starts the application under the gdb debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 114 for more information.
eadb	Starts the application under adb—the absolute debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 114 for more information.
ewdb	Starts the application under the wdb debugger. The debugger must be in the command search path. See “Debugging HP MPI applications” on page 114 for more information.
l	Reports memory leaks caused by not freeing memory allocated when an HP MPI job is run. For example, when you create a new communicator or user-defined datatype after you call MPI_Init, you must free the memory allocated to these objects before you call MPI_Finalize. In C, this is analogous to making calls to malloc() and free() for each object created during program execution. Setting the l option may decrease application performance.

f

Forces MPI errors to be fatal. Using the f option sets the MPI_ERRORS_ARE_FATAL error handler, ignoring the programmer's choice of error handlers. This option can help you detect nondeterministic error problems in your code.

If your code has a customized error handler that does not report that an MPI call failed, you will not know that a failure occurred. Thus your application could be catching an error with a user-written error handler (or with MPI_ERRORS_RETURN) which masks a problem.

`s[a|p][#]`

Selects signal and maximum time delay for guaranteed message progression. The `sa` option selects `SIGALRM`. The `sp` option selects `SIGPROF`. The # option is the number of seconds to wait before issuing a signal to trigger message progression. The default value for the MPI library is `sp604800`, which issues a `SIGPROF` once a week. If the application uses both signals for its own purposes, you must disable the heart-beat signals. A time value of zero seconds disables the heart beats.

This mechanism is used to guarantee message progression in applications that use nonblocking messaging requests followed by prolonged periods of time in which HP MPI routines are not called.

Generating a UNIX signal introduces a performance penalty every time the application processes are interrupted. As a result, while some applications will benefit from it, others may experience a decrease in performance. As part of tuning the performance of an application, you can control the behavior of the heart-beat signals by changing their time period or by turning them off. This is accomplished by setting the time period of the `s` option in the MPI_FLAGS environment variable (for example: s600). Time is in seconds.

You can use the `s[a][p]#` option with the thread-compliant library as well as the standard non thread-compliant library. Setting `s[a][p]#` for the thread-compliant library has the same effect as setting `MPI_MT_FLAGS=ct` when you use a value greater than

0 for #. The default value for the thread-compliant library is `sp0`. `MPI_MT_FLAGS=ct` takes priority over the default `MPI_FLAGS=sp0`.

Refer to “MPI_MT_FLAGS” on page 44 and “Thread-compliant library” on page 170 for additional information.

`y[#]` Enables spin-yield logic. # is the spin value and is an integer between zero and 10,000. The spin value specifies the number of milliseconds a process should block waiting for a message before yielding the CPU to another process.

How you apply spin-yield logic depends on how well synchronized your processes are. For example, if you have a process that wastes CPU time blocked, waiting for messages, you can use spin-yield to ensure that the process relinquishes the CPU to other processes. Do this in your appfile, by setting `y[#]` to `y0` for the process in question. This specifies zero milliseconds of spin (that is, immediate yield).

On the other extreme, you can set spin-yield for a process so that it spins continuously, that is, it does not relinquish the CPU while it waits for a message. To spin without yielding, specify y without a spin value.

If the time a process is blocked waiting for messages is short, you can possibly improve performance by setting a spin value (between 0 and 10,000,) that ensures the process does not relinquish the CPU until after the message is received, thereby reducing latency.

The system treats a nonzero spin value as a recommendation only. It does not guarantee that the value you specify is used.

Refer to “Appfiles” on page 55 for details about how to create an appfile and assign ranks.

`o` Writes an optimization report to stdout. `MPI_Cart_create` and `MPI_Graph_create` optimize the mapping of processes onto the virtual topology if rank reordering is enabled.

+E2	Sets -1 as the value of .TRUE. and 0 as the value for FALSE. when returning logical values from HP MPI routines called within Fortran 77 applications.
D	Dumps shared memory configuration information. Use this option to get shared memory values that are useful when you want to set the MPI_SHMCNTL flag.
E	Disables function parameter error checking. Turning off argument checking can improve performance.
z	Enables zero-buffering mode. Set this flag to convert MPI_Send and MPI_Rsend calls in your code to MPI_Ssend, without rewriting your code. Refer to Troubleshooting, "Application hangs in MPI_Send" on page 130, for information about how using this option can help uncover nonportable code in your MPI application.

MP_GANG

MP_GANG enables gang scheduling. Gang scheduling improves the latency for synchronization by ensuring that all runable processes in a gang are scheduled simultaneously. Processes waiting at a barrier, for example, do not have to wait for processes that are not currently scheduled. This proves most beneficial for applications with frequent synchronization operations. Applications with infrequent synchronization, however, may perform better if gang scheduling is disabled.

Process priorities for gangs are managed identically to timeshare policies. The timeshare priority scheduler determines when to schedule a gang for execution. While it is likely that scheduling a gang will preempt one or more higher priority timeshare processes, the gang-schedule policy is fair overall. In addition, gangs are scheduled for a single time slice, which is the same for all processes in the system.

MPI processes are allocated statically at the beginning of execution. As an MPI process creates new threads, they are all added to the same gang if MP_GANG is enabled. The MP_GANG syntax is as follows:

[ON|OFF]

where

ON	Enables gang scheduling.

OFF	Disables gang scheduling.

For multihost configurations, you need to set MP_GANG for each appfile entry. Refer to the -e option in “Creating an appfile” on page 55.

You can also use the HP-UX utility mpsched(1) to enable gang scheduling. Refer to the HP-UX gang_sched(7) and mpsched(1) manpages for more information.

MPI_GLOBMEMSIZE

MPI_GLOBMEMSIZE specifies the amount of shared memory allocated for all processes in an HP MPI application. The MPI_GLOBMEMSIZE syntax is as follows:

amount

where *amount* specifies the total amount of shared memory in bytes for all processes. The default is 2 Mbytes for up to 64-way applications and 4 Mbytes for larger applications.

Be sure that the value specified for MPI_GLOBMEMSIZE is less than the amount of global shared memory allocated for the host. Otherwise, swapping overhead will degrade application performance.

MPI_INSTR

MPI_INSTR enables counter instrumentation for profiling HP MPI applications. The MPI_INSTR syntax is a colon-separated list (no spaces between options) as follows:

prefix[:*b#1*,*#2*[:*b#1*,*#2*][...]][:nd][:nc][:off][:nl][:np][:nm][:c]

where

prefix	Specifies the instrumentation output file prefix. The rank zero process writes the application's measurement data to *prefix*.instr in ASCII, and to *prefix*.mpiview in a graphical format readable by mpiview. If the prefix does not represent an absolute pathname, the instrumentation output file is opened in the working directory of the rank zero process when MPI_Init is called.
b#1,#2	Redefines the instrumentation message bins to include a bin having byte range *#1* and *#2* inclusive. The high bound of the range (#2) can be infinity, representing

	the largest possible message size. When you specify a number of bin ranges, ensure that the ranges do not overlap.
`nd`	Disables rank-by-peer density information when running counter instrumentation.
`nc`	Specifies no clobber. If the instrumentation output file exists, `MPI_Init` aborts.
`off`	Specifies counter instrumentation is initially turned off and only begins after all processes collectively call `MPIHP_Trace_on`.
`nl`	Specifies not to dump a long breakdown of the measurement data to the instrumentation output file (that is, do not dump minimum, maximum, and average time data).
`np`	Specifies that a per-process breakdown of the measurement data is not dumped to the instrumentation output file.
`nm`	Specifies that message-size measurement data is not dumped to the instrumentation output file.
`c`	Specifies that time measurement data is not dumped to the instrumentation output file.

Refer to “Using counter instrumentation” on page 68 for more information.

Even though you can specify profiling options through the `MPI_INSTR` environment variable, the recommended approach is to use the `mpirun` command with the `-i` option instead. Using `mpirun` to specify profiling options guarantees that multihost applications do profiling in a consistent manner. Refer to “mpirun” on page 49 for more information.

Counter instrumentation and trace-file generation (used in conjunction with XMPI) are mutually exclusive profiling techniques.

NOTE When you enable instrumentation for multihost runs, and invoke mpirun either on a host where at least one MPI process is running, or on a host remote from all your MPI processes, HP MPI writes the instrumentation output files (*prefix*.instr and *prefix*.mpiview) to the working directory on the host that is running rank 0.

MPI_LOCALIP

`MPI_LOCALIP` specifies the host IP address that is assigned throughout a session. Ordinarily, mpirun and XMPI determine the IP address of the host they are running on by calling `gethostbyaddr`. However, when a host uses a SLIP or PPP protocol, the host's IP address is dynamically assigned only when the network connection is established. In this case, `gethostbyaddr` may not return the correct IP address.

The `MPI_LOCALIP` syntax is as follows:

xxx.*xxx*.*xxx*.*xxx*

where *xxx*.*xxx*.*xxx*.*xxx* specifies the host IP address.

MPI_MT_FLAGS

MPI_MT_FLAGS controls runtime options when you use the thread-compliant version of HP MPI. The MPI_MT_FLAGS syntax is a comma separated list as follows:

```
[ct,][single,][fun,][serial,][mult]
```

where

ct	Creates a hidden communication thread for each rank in the job. When you enable this option, be careful not to oversubscribe your system. For example, if you enable ct for a 16-process application running on a 16-way machine, the result will be a 32-way job.
single	Asserts that only one thread executes.
fun	Asserts that a process can be multithreaded, but only the main thread makes MPI calls (that is, all calls are funneled to the main thread).
serial	Asserts that a process can be multithreaded, and multiple threads can make MPI calls, but calls are serialized (that is, only one call is made at a time).
mult	Asserts that multiple threads can call MPI at any time with no restrictions.

Setting MPI_MT_FLAGS=ct has the same effect as setting MPI_FLAGS=s[a][p]#, when the value of # that is greater than 0. MPI_MT_FLAGS=ct takes priority over the default MPI_FLAGS=sp0 setting. Refer to “MPI_FLAGS” on page 37.

The single, fun, serial, and mult options are mutually exclusive. For example, if you specify the serial and mult options in MPI_MT_FLAGS, only the last option specified is processed (in this case, the mult option). If no runtime option is specified, the default is mult.

For more information about using MPI_MT_FLAGS with the thread-compliant library, refer to “Thread-compliant library” on page 170.

MPI_NOBACKTRACE

On PA-RISC systems, a stack trace is printed when the following signals occur within an application:

- SIGILL
- SIGBUS
- SIGSEGV
- SIGSYS

In the event one of these signals is not caught by a user signal handler, HP MPI will display a brief stack trace that can be used to locate the signal in the code.

```
Signal 10: bus error
PROCEDURE TRACEBACK:

(0)   0x0000489c   bar + 0xc   [./a.out]
(1)   0x000048c4   foo + 0x1c   [,/,/a.out]
(2)   0x000049d4   main + 0xa4   [./a.out]
(3)   0xc013750c   _start + 0xa8   [/usr/lib/libc.2]
(4)   0x0003b50    $START$ + 0x1a0 [./a.out]
```

This feature can be disabled for an individual signal handler by declaring a user-level signal handler for the signal. To disable for all signals, set the environment variable `MPI_NOBACKTRACE`:

```
% setenv MPI_NOBACKTRACE
```

See “Backtrace functionality” on page 119 for more information.

MPI_REMSH

`MPI_REMSH` specifies a command other than the default `remsh` to start remote processes. The mpirun, mpijob, and mpiclean utilities support `MPI_REMSH`. For example, you can set the environment variable to use a secure shell:

```
% setenv MPI_REMSH /bin/ssh
```

The alternative remote shell command should be a drop-in replacement for /usr/bin/remsh, that is, the argument syntax for the alternative shell should be the same as for /usr/bin/remsh.

MPI_SHMEMCNTL

MPI_SHMEMCNTL controls the subdivision of each process's shared memory for the purposes of point-to-point and collective communications. The MPI_SHMEMCNTL syntax is a comma separated list as follows:

nenv, frag, generic

where

nenv	Specifies the number of envelopes per process pair. The default is 8.
frag	Denotes the size in bytes of the message-passing fragments region. The default is 87.5 percent of shared memory after mailbox and envelope allocation.
generic	Specifies the size in bytes of the generic-shared memory region. The default is 12.5 percent of shared memory after mailbox and envelope allocation.

MPI_TMPDIR

By default, HP MPI uses the /tmp directory to store temporary files needed for its operations. MPI_TMPDIR is used to point to a different temporary directory. The MPI_TMPDIR syntax is

directory

where *directory* specifies an existing directory used to store temporary files.

MPI_WORKDIR

By default, HP MPI applications execute in the directory where they are started. MPI_WORKDIR changes the execution directory. The MPI_WORKDIR syntax is shown below:

directory

where *directory* specifies an existing directory where you want the application to execute.

MPI_XMPI

MPI_XMPI specifies options for runtime trace generation. These options represent an alternate way to set tracing rather than using the trace options supplied with mpirun.

The argument list for MPI_XMPI contains the prefix name for the file where each process writes its own trace data.

Before your application exits, MPI_Finalize consolidates the process trace files to a single trace file, named *prefix*.tr. If the file prefix does not represent an absolute pathname (for example, /tmp/test), the consolidated trace file is stored in the directory in which the process is executing MPI_Init.

The MPI_XMPI syntax is a colon-separated list (no spaces between options) as follows:

prefix[:*bs###*][:nc][:off][:s]

where

prefix	Specifies the tracing output file prefix. *prefix* is a required parameter.
bs###	Denotes the buffering size in kbytes for dumping raw trace data. Actual buffering size may be rounded up by the system. The default buffering size is 4096 kbytes. Specifying a large buffering size reduces the need to flush raw trace data to a file when process buffers reach capacity. Flushing too frequently can cause communication routines to run slower.
nc	Specifies no clobber, which means that an HP MPI application aborts if a file with the name specified in *prefix* already exists.
off	Denotes that trace generation is initially turned off and only begins after all processes collectively call MPIHP_Trace_on.
s	Specifies a simpler tracing mode by omitting tracing for MPI_Test, MPI_Testall, MPI_Testany, and MPI_Testsome calls that do not complete a request. This option may reduce the size of trace data so that xmpi runs faster.

Even though you can specify tracing options through the `MPI_XMPI` environment variable, the recommended approach is to use the `mpirun` command with the `-t` option instead. In this case, the specifications you provide with the `-t` option take precedence over any specifications you may have set with `MPI_XMPI`. Using `mpirun` to specify tracing options guarantees that multihost applications do tracing in a consistent manner. Refer to "mpirun" on page 49 for more information.

Trace-file generation (in conjunction with XMPI) and counter instrumentation are mutually exclusive profiling techniques.

NOTE To generate tracing output files for multihost applications, you must invoke mpirun on a host where at least one MPI process is running. HP MPI writes the trace file (*prefix*.tr) to the working directory on the host where mpirun runs.

When you enable tracing for multihost runs, and invoke mpirun on a machine that is not running an MPI process, HP MPI issues a warning and does not write the trace file.

TOTALVIEW

When you use the TotalView debugger, HP MPI uses your PATH variable to find TotalView. You can also set the absolute path and TotalView specific options in the TOTALVIEW environment variable. This environment variable is used by mpirun.

`setenv TOTALVIEW` */opt/totalview/bin/totalview* [*totalview_options*]

Runtime utility commands

HP MPI provides a set of utility commands to supplement the MPI library routines. These commands are listed below and described in the following sections:

- `mpirun`

  This section also includes discussion of Shared library support, Appfiles, the Multipurpose daemon process, and Generating multihost instrumentation profiles.

- `mpijob`
- `mpiclean`
- `xmpi`
- `mpiview`

`mpirun`

The new HP MPI 1.7 start-up provides the following advantages:

- Provides support for shared libraries
- Allows many multi-threaded applications to use high-performance single-threaded code paths
- Includes a cleaner tear-down mechanism for abnormal termination
- Provides a simplified path to provide bug fixes to the field

CAUTION **HP MPI 1.7 is backward-compatible at a source-code level only. It is not start-up backward-compatible. Your previous version of HP MPI must be retained in order to run executables built with archive libraries on previous versions of HP MPI.**

The new HP MPI 1.7 start-up requires that MPI be installed in the same directory on every execution host. The default is the location from which `mpirun` is executed. This can be overridden with the `MPI_ROOT` environment variable. We recommend setting the `MPI_ROOT` environment variable prior to starting `mpirun`.

NOTE Options `-w` and `-W` are no longer supported.

Previous versions of HP MPI allowed `mpirun` to exit prior to application termination by specifying the `-W` option. Because the `-W` option used with `mpirun` is no longer supported, place `mpirun` in the background to achieve similar functionality.

`mpirun` syntax has four formats:

- For applications where all processes execute the same program on the same host:

  `mpirun [-np #] [-help] [-version] [-djpv] [-ck] [-t` *spec*`] [-i` *spec*`] [-h` *host*`] [-l` *user*`] [-e` *var*`[=`*val*`]]... [-sp` *paths*`] [-tv]` *program* `[`*args*`]`

  For example:

  % **`mpirun -j -np 3 send_receive`**

  runs the send_receive application with three processes and prints out the job ID.

- For applications that consist of multiple programs or that run on multiple hosts:

  `mpirun [-help] [-version] [-djpv] [-ck] [-t` *spec*`] [-i` *spec*`] [-commd] [-tv] -f` *appfile* `[--` *extra_args_for_appfile*`]`

  In this case, each program in the application is listed in a file called an appfile. Refer to “Appfiles” on page 55 for more information.

  For example:

  % **`mpirun -t my_trace -f my_appfile`**

  enables tracing, specifies the prefix for the tracing output file is my_trace, and runs an appfile named my_appfile.

- To invoke LSF for applications where all processes execute the same program on the same host:

  `bsub [`*lsf_options*`] pam -mpi mpirun [`*mpirun_options*`]` *program* `[`*args*`]`

  In this case, LSF assigns a host to the MPI job.

For example:

% **bsub pam -mpi mpirun -np 4 compute_pi**

requests a host assignment from LSF and runs the compute_pi application with four processes. Refer to "Assigning hosts using LSF" on page 64 for more information.

NOTE This is the last release of HP MPI that will support tightly-coupled integration between LSF's Parallel Application Manager (PAM) and HP MPI. Shell scripts will be provided to enable similar functionality when support for this feature is discontinued.

- To invoke LSF for applications that run on multiple hosts:

  `bsub` [*lsf_options*] `pam -mpi mpirun` [*mpirun_options*] `-f` *appfile* [`--` *extra_args_for_appfile*]

  In this case, each host specified in the appfile is treated as a symbolic name, referring to the host that LSF assigns to the MPI job.

  For example:

  % **bsub pam -mpi mpirun -f my_appfile**

  runs an appfile named my_appfile and requests host assignments for all remote and local hosts specified in my_appfile. If my_appfile contains the following items:

  ```
  -h voyager -np 10 send_receive
  -h enterprise -np 8 compute_pi
  ```

  Host assignments are returned for the two symbolic links voyager and enterprise.

  When requesting a host from LSF, you must ensure that the path to your executable file is accessible by all machines in the resource pool. Refer to "Assigning hosts using LSF" on page 64 for more information.

where [*mpirun_options*] are:

`-ck`

Behaves like the `-p` option, but supports two additional checks of your MPI application; it checks if the specified host machines and programs are available, and also checks for access or permission problems.

`-commd`

Routes all off-host communication through daemons rather than between processes. Refer to “Communicating using daemons” on page 62 for more information.

`-d`

Turns on debug mode.

`-e` *var*`[=`*val*`]`

Sets the environment variable *var* for the program and gives it the value *val* if provided. Environment variable substitutions (for example, $FOO) are supported in the *val* argument.

`-f` *appfile*

Specifies the *appfile* that mpirun parses to get program and process count information for the run. Refer to“Creating an appfile” on page 55 for details about setting up your appfile.

`-h` *host*

Specifies a *host* on which to start the processes (default is *local_host*).

`-help`

Prints usage information for the utility.

`-i` *spec*

Enables runtime instrumentation profiling for all processes. *spec* specifies options used when profiling. The options are the same as those for the environment variable `MPI_INSTR`. For example, the following is a valid command line:

```
% mpirun -i mytrace:nd:nc -f appfile
```

Refer to “MPI_INSTR” on page 41 for an explanation of `-i` options.

`-j`

Prints the HP MPI job ID.

`-l` *user*

Specifies the username on the target host (default is local username).

`-np` *#*

Specifies the number of processes to run.

`-p`

Turns on pretend mode. That is, the system goes through the motions of starting an HP MPI application but does not create processes. This is useful for debugging and checking whether the appfile is set up correctly.

`-sp` *paths*

Sets the target shell PATH environment variable to *paths*. Search paths are separated by a colon.

`-t` *spec*

Enables runtime trace generation for all processes. *spec* specifies options used when tracing. The options are the same as those for the environment variable `MPI_XMPI`. For example, the following is a valid command line:

```
% mpirun -t mytrace:off:nc -f appfile
```

Refer to "MPI_XMPI" on page 47 for an explanation of `-t` options.

`-tv`

Specifies that the application runs with the TotalView debugger. This option is not supported when you run mpirun under LSF.

`-v`

Turns on verbose mode.

`-version`

Prints the version information.

args

Specifies command-line arguments to the program—A space separated list of arguments.

`--` *extra_args_for_appfile*

Specifies extra arguments to be applied to the programs listed in the appfile—A space separated list of arguments. Use this option at the end of your command line to append extra arguments to each line of your appfile. Refer to the example in "Adding program arguments to your appfile" on page 56 for details.

program

Specifies the name of the executable file to run.

IMPI_options

Specifies this mpirun is an IMPI client. Refer to "IMPI" on page 64 for more information on IMPI, as well as a complete list of IMPI options.

lsf_options

Specifies bsub options that the load-sharing facility (LSF) applies to the entire job (that is, every host). Refer to the bsub(1) man page for a list of options you can use. Note that LSF must be installed for *lsf_options* to work correctly.

`-stdio=`[*options*]

Specifies standard IO options. Refer to "External input and output" on page 126 for more information on standard IO, as well as a complete list of stdio options.

CAUTION **The `-help`, `-version`, `-p` , and `-tv` options are not supported with the `bsub pam -mpi mpirun` startup method.**

Shared library support

When a library is shared, programs using it contain only references to library routines, as opposed to archive libraries which must be linked into every program using them. The same copy of the shared library is referenced by each executable using it.

You can use HP MPI 1.7 as archive or shared libraries. However, your previous version of HP MPI must be retained in order to run executables built with archive libraries on previous versions of HP MPI.

An advantage of shared libraries is that when the library is updated (e.g. to fix a bug) all programs which use the library immediately enjoy the fix. The disk and memory savings of shared libraries is offset by a slight performance penalty when a shared executable starts up. References to shared library routines must be resolved by finding the libraries containing those routines. However, references need be resolved only once, so the performance penalty is quite small.

In order to use shared libraries, HP MPI must be installed on all machines in the same directory.

Shared libraries are used by default. In order to link with archive libraries, use the `-aarchive_shared` linker option. Archive libraries are not available on the Itanium-based version of HP MPI.

Appfiles

An appfile is a text file that contains process counts and a list of programs. When you invoke `mpirun` with the name of the appfile, `mpirun` parses the appfile to get information for the run. You can use an appfile when you run a single executable file on a single host, and you must use an appfile when you run on multiple hosts or run multiple executable files.

Creating an appfile

The format of entries in an appfile is line oriented. Lines that end with the backslash (\) character are continued on the next line, forming a single logical line. A logical line starting with the pound (#) character is treated as a comment. Each program, along with its arguments, is listed on a separate logical line.

The general form of an appfile entry is:

`[-h` *remote_host*`] [-e` *var*`[=`*val*`] [...]] [-l` *user*`] [-sp` *paths*`] [-np` *#*`]` *program* `[`*args*`]`

where

`-h` *remote_host*	Specifies the remote host where a remote executable file is stored. The default is to search the local host. *remote_host* is either a host name or an IP address.
`-e` *var*=*val*	Sets the environment variable *var* for the program and gives it the value *val*. The default is not to set environment variables. When you use `-e` with the `-h` option, the environment variable is set to *val* on the remote host.
`-l` *user*	Specifies the user name on the target host. The default is the current user name.
`-sp` *paths*	Sets the target shell PATH environment variable to *paths*. Search paths are separated by a colon.
`-np` *#*	Specifies the number of processes to run. The default value for # is 1.
program	Specifies the name of the executable to run. `mpirun` searches for the executable in the paths defined in the PATH environment variable.

args Specifies command line arguments to the program. Options following a program name in your appfile are treated as program arguments and are not processed by mpirun.

Adding program arguments to your appfile

When you invoke mpirun using an appfile, arguments for your program are supplied on each line of your appfile—Refer to "Creating an appfile" on page 55. HP MPI also provides an option on your mpirun command line to provide additional program arguments to those in your appfile. This is useful if you wish to specify extra arguments for each program listed in your appfile, but do not wish to edit your appfile.

To use an appfile when you invoke mpirun, use one of the following as described in "mpirun" on page 49:

- `mpirun [`*`mpirun_options`*`] -f` *appfile* `[--` *extra_args_for_appfile*`]`
- `bsub [`*`lsf_options`*`] pam -mpi mpirun [`*`mpirun_options`*`] -f` *appfile* `[--` *extra_args_for_appfile*`]`

The `--` *extra_args_for_appfile* option is placed at the end of your command line, after *appfile*, to add options to each line of your appfile.

CAUTION **Arguments placed after - - are treated as program arguments, and are not processed by mpirun. Use this option when you want to specify program arguments for each line of the appfile, but want to avoid editing the appfile.**

For example, suppose your appfile contains

```
-h voyager -np 10 send_receive arg1 arg2
-h enterprise -np 8 compute_pi
```

If you invoke mpirun using the following command line:

`mpirun -f` *appfile* `--` *arg3 -arg4 arg5*

- The `send_receive` command line for machine voyager becomes:

  ```
  send_receive arg1 arg2 arg3 -arg4 arg5
  ```

- The `compute_pi` command line for machine enterprise becomes:

  ```
  compute_pi arg3 -arg4 arg5
  ```

When you use the `--` *extra_args_for_appfile* option, it must be specified at the end of the `mpirun` command line.

Setting remote environment variables

To set environment variables on remote hosts use the `-e` option in the appfile. For example, to set the variable MPI_FLAGS:

`-h` *remote_host* `-e MPI_FLAGS=`*val* `[-np` *#*`]` *program* `[`*args*`]`

Assigning ranks and improving communication

The ranks of the processes in MPI_COMM_WORLD are assigned and sequentially ordered according to the order the programs appear in the appfile.

For example, if your appfile contains

```
-h voyager -np 10 send_receive
-h enterprise -np 8 compute_pi
```

HP MPI assigns ranks 0 through 9 to the 10 processes running send_receive and ranks 10 through 17 to the 8 processes running compute_pi.

You can use this sequential ordering of process ranks to your advantage when you optimize for performance on multihost systems. You can split process groups according to communication patterns to reduce or remove interhost communication hot spots.

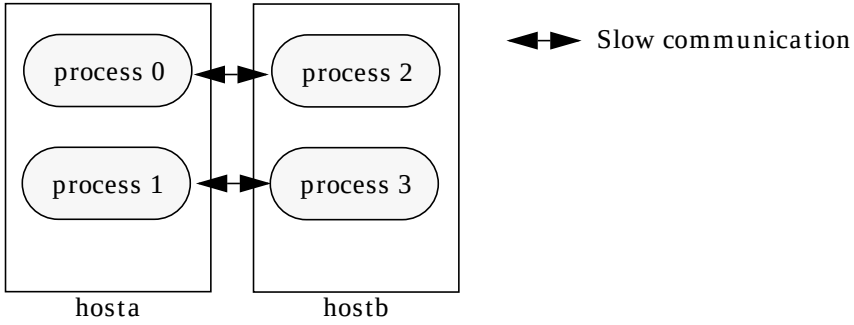
For example, if you have the following:

- A multi-host run of four processes
- Two processes per host on two hosts
- Communication between ranks 0—2 and 1—3 is slow (you can identify communication hot spots using HP MPI's instrumentation; refer to "mpiview" on page 62)

You could use an appfile that contains the following:

```
-h hosta -np 2 program1
-h hostb -np 2 program2
```

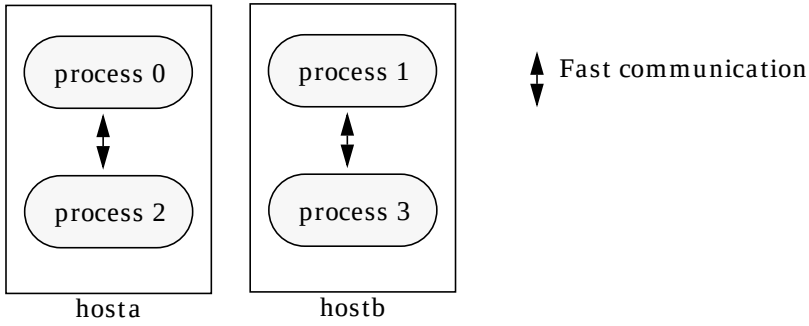
However, this places processes 0 and 1 on hosta and processes 2 and 3 on hostb, resulting in interhost communication between the ranks identified as having slow communication:



A more optimal appfile for this example would be

```
-h hosta -np 1 program1
-h hostb -np 1 program2
-h hosta -np 1 program1
-h hostb -np 1 program2
```

This places ranks 0 and 2 on hosta and ranks 1 and 3 on hostb. This placement allows intrahost communication between ranks that are identified as communication hot spots. Intrahost communication yields better performance than interhost communication.



Multipurpose daemon process

HP MPI incorporates a multipurpose daemon process that provides start–up, communication, and termination services. The daemon operation is transparent. HP MPI sets up one daemon per host (or appfile entry) for communication. Refer to “Communicating using daemons” on page 62 for daemon details.

NOTE Because HP MPI sets up one daemon per host (or appfile entry) for communication, when you invoke your application with -np x, HP MPI generates x+1 processes.

Generating multihost instrumentation profiles

To generate tracing output files for multihost applications, you must invoke mpirun on a host where at least one MPI process is running. HP MPI writes the trace file (*prefix*.tr) to the working directory on the host where mpirun runs.

When you enable instrumentation for multihost runs, and invoke mpirun either on a host where at least one MPI process is running, or on a host remote from all your MPI processes, HP MPI writes the instrumentation output files (*prefix*.instr and *prefix*.mpiview) to the working directory on the host that is running rank 0.

mpijob

`mpijob` lists the HP MPI jobs running on the system. Invoke `mpijob` on the same host as you initiated `mpirun`. `mpijob` syntax is shown below:

`mpijob [-help] [-a] [-u] [-j` *id* `[`*id id ...*`]]`

where

`-help`	Prints usage information for the utility.
`-a`	Lists jobs for all users.
`-u`	Sorts jobs by user name.
`-j` *id*	Provides process status for job *id*. You can list a number of job IDs in a space-separated list.

When you invoke `mpijob`, it reports the following information for each job:

JOB	HP MPI job identifier.
USER	User name of the owner.
NPROCS	Number of processes.
PROGNAME	Program names used in the HP MPI application.

By default, your jobs are listed by job ID in increasing order. However, you can specify the `-a` and `-u` options to change the default behavior.

An `mpijob` output using the `-a` and `-u` options is shown below listing jobs for all users and sorting them by user name.

```
JOB      USER      NPROCS   PROGNAME
22623    charlie   12       /home/watts
22573    keith     14       /home/richards
22617    mick      100      /home/jagger
22677    ron        4        /home/wood
```

When you specify the `-j` option, `mpijob` reports the following for each job:

RANK	Rank for each process in the job.
HOST	Host where the job is running.
PID	Process identifier for each process in the job.
LIVE	Indicates whether the process is running (an x is used) or has been terminated.
PROGNAME	Program names used in the HP MPI application.

mpiclean

mpiclean kills processes in an HP MPI application. Invoke mpiclean on the host on which you initiated mpirun.

The MPI library checks for abnormal termination of processes while your application is running. In some cases, application bugs can cause processes to deadlock and linger in the system. When this occurs, you can use mpijob to identify hung jobs and mpiclean to kill all processes in the hung application.

mpiclean syntax has two forms:

1. `mpiclean [-help] [-v] -j` *id* `[`*id id* `....]`

2. `mpiclean [-help] [-v] -m`

where

`-help`	Prints usage information for the utility.
`-v`	Turns on verbose mode.
`-m`	Cleans up your shared-memory segments.
`-j` *id*	Kills the processes of job number *id*. You can specify multiple job IDs in a space-separated list. Obtain the job ID using the `-j` option when you invoke `mpirun`.

The first syntax is used for all servers and is the preferred method to kill an MPI application. You can only kill jobs that are your own.

The second syntax is used when an application aborts during `MPI_Init`, and the termination of processes does not destroy the allocated shared-memory segments.

xmpi

xmpi invokes the XMPI utility, an X/Motif graphical user interface for running applications, monitoring processes and messages, and viewing trace files. The xmpi syntax is shown below:

`xmpi [-h][-bg` *arg*`][-bd` *arg*`][-bw` *arg*`][-display` *arg*`]`
`[-fg` *arg*`][-geometry` *arg*`][-iconic][-title` *arg*`]`

where the xmpi arguments are standard X/Motif arguments.

The X resource settings that determine the default settings for displaying XMPI are in /opt/mpi/lib/X11/app-defaults/XMPI. Refer to "Using XMPI" on page 78 and Appendix B, "XMPI resource file" for more information.

NOTE HP MPI 1.7 is the last release that supports XMPI and mpiview.

XMPI and mipview are not supported for Itanium-based systems.

`mpiview`

`mpiview` invokes the mpiview utility, a graphical user interface to display counter instrumentation data. `mpiview` reads a *prefix*.mpiview file containing the counter instrumentation data. You specified the filename *prefix* either in the environment variable MPI_INSTR (refer to "MPI_INSTR" on page 41) or by using the `-i` option with the `mpirun` command (refer to "mpirun" on page 49). For example,

```
% mpiview my_data.mpiview
```

invokes mpiview and displays counter instrumentation data from the my_data.mpiview file.

For more information, refer to "Creating an instrumentation profile" on page 68 and "Viewing instrumentation data with mpiview" on page 73.

NOTE HP MPI 1.7 is the last release that supports XMPI and mpiview.

XMPI and mipview are not supported for Itanium-based systems.

Communicating using daemons

By default, off-host communication between processes is implemented using direct socket connections between process pairs. For example, if process A on host1 communicates with processes D and E on host2, then process A sends messages using a separate socket for each process D and E.

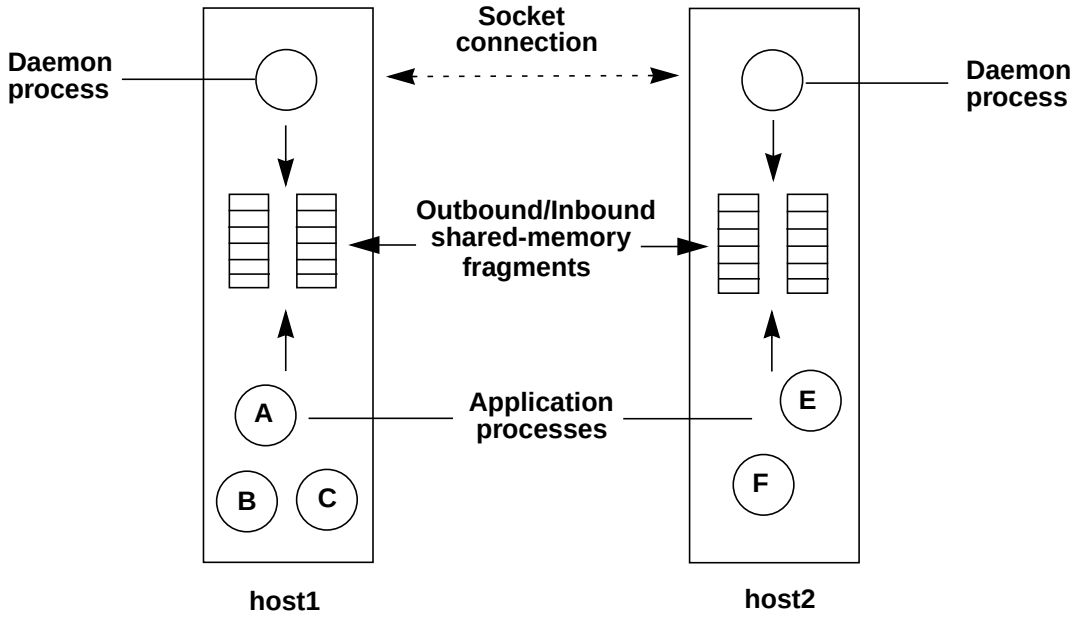
This is referred to as the n-squared or direct approach because to run an *n*-process application, n^2 sockets are required to allow processes on one host to communicate with processes on other hosts. When you use this direct approach, you should be careful that the total number of open sockets does not exceed the system limit.

You can also use an indirect approach and specify that all off-host communication occur between daemons, by specifying the `-commd` option to the mpirun command. In this case, the processes on a host use shared

memory to send messages to and receive messages from the daemon. The daemon, in turn, uses a socket connection to communicate with daemons on other hosts.

Figure 1 shows the structure for daemon communication.

Figure 1 Daemon communication



To use daemon communication, specify the `-commd` option in the `mpirun` command. Once you have set the `-commd` option, you can use the `MPI_COMMD` environment variable to specify the number of shared-memory fragments used for inbound and outbound messages. Refer to “mpirun” on page 49 and “MPI_COMMD” on page 35 for more information.

Daemon communication can result in lower application performance. Therefore, use it only when scaling an application to a large number of hosts.

NOTE HP MPI sets up one daemon per host (or appfile entry) for communication. If you invoke your application with `-np x`, HP MPI generates x+1 processes.

IMPI

The Interoperable MPI protocol (IMPI) extends the power of MPI by allowing applications to run on heterogeneous clusters of machines with various architectures and operating systems, while allowing the program to use a different implementation of MPI on each machine.

This is accomplished without requiring any modifications to the existing MPI specification. That is, IMPI does not add, remove, or modify the semantics of any of the existing MPI routines. All current valid MPI programs can be run in this way without any changes to their source code.

In IMPI, all messages going out of a host go through the daemon. The messages between daemons have the fixed message format. The protocols in different IMPI implementations are the same.

Currently, IMPI is not supported in multi-threaded library. If the user application is a multi-threaded program, it is not allowed to start as an IMPI job.

An IMPI server is available for download from Notre Dame at: http://www.lsc.nd.edu/research/impi

The IMPI syntax is:

`mpirun` [*-client # ip:port*]

where

`-client`	Specifies this mpirun is an IMPI client.
#	Specifies the client number. The first # is 0.
`ip`	Specifies the IP address of the IMPI server.
`port`	Specifies the port number of the IMPI server.

Assigning hosts using LSF

The load-sharing facility (LSF) allocates one or more hosts to run an MPI job. In general, LSF improves resource utilization for MPI jobs that run in multihost environments. LSF handles the job scheduling and the allocation of the necessary hosts and HP MPI handles the task of starting up the application's processes on the hosts selected by LSF.

By default mpirun starts the MPI processes on the hosts specified by the user, in effect handling the direct mapping of host names to IP addresses. When you use LSF to start MPI applications, the host names, specified to mpirun or implicit when the -h option is not used, are treated as symbolic variables that refer to the IP addresses that LSF assigns. Use LSF to do this mapping by specifying a variant of `mpirun` to execute your job.

NOTE This is the last release of HP MPI that will support tightly-coupled integration between LSF's Parallel Application Manager (PAM) and HP MPI. Shell scripts will be provided to enable similar functionality when support for this feature is discontinued.

Native Language Support

By default, diagnostic messages and other feedback from HP MPI are provided in English. Support for other languages is available through the use of the Native Language Support (NLS) catalog and the internationalization environment variable NLSPATH.

The default NLS search path for HP MPI is $NLSPATH. Refer to the environ(5) man page for NLSPATH usage.

When an MPI language catalog is available, it represents HP MPI messages in two languages. The messages are paired so that the first in the pair is always the English version of a message and the second in the pair is the corresponding translation to the language of choice.

Refer to the hpnls (5), environ (5), and lang (5) man pages for more information about Native Language Support.

4 Profiling

This chapter provides information about utilities you can use to analyze HP MPI applications. The topics covered are:

- Using counter instrumentation
  - Creating an instrumentation profile
  - Viewing ASCII instrumentation data
  - Viewing instrumentation data with mpiview
- Using XMPI
  - Working with postmortem mode
  - Working with interactive mode
- Using CXperf
- Using the profiling interface

Using counter instrumentation

Counter instrumentation is a lightweight method for generating cumulative runtime statistics for your MPI applications. When you create an instrumentation profile, HP MPI creates two file formats, an ASCII format, and a graphical format readable by the mpiview utility.

You can create instrumentation profiles for applications linked with the standard HP MPI library, and for applications linked with HP MPI version 1.7, you can also create profiles for applications linked with the thread-compliant library. Instrumentation is not supported for applications linked with the diagnostic library (-ldmpi).

Creating an instrumentation profile

Create an instrumentation profile using one of the following methods:

- Use the following syntax:

  `mpirun -i` *spec* `-np #` *program*

  Refer to “Preparing mpiview instrumentation files” on page 23 and “mpirun” on page 49 for more details about implementation and syntax.

  For example, to create an instrumentation profile for an application called compute_pi.f, enter:

  `%` **`mpirun -i compute_pi -np 2 compute_pi`**

  This invocation creates an instrumentation profile in two formats: compute_pi.instr (ASCII) and compute_pi.mpiview (graphical).

- Specify a filename prefix using the MPI_INSTR environment variable. Refer to “MPI_INSTR” on page 41 for syntax information. For example,

  `%` **`setenv MPI_INSTR`** ***`compute_pi`***

  Specifies the instrumentation output file prefix as compute_pi.

Specifications you make using `mpirun -i` override any specifications you make using the `MPI_INSTR` environment variable.

MPIHP_Trace_on and MPIHP_Trace_off

By default, the entire application is profiled from `MPI_Init` to `MPI_Finalize`. However, HP MPI provides the nonstandard `MPIHP_Trace_on` and `MPIHP_Trace_off` routines to collect profile information for selected code sections only.

To use this functionality:

1. Insert the `MPIHP_Trace_on` and `MPIHP_Trace_off` pair around code that you want to profile.
2. Build the application and invoke `mpirun` with the `-i off` option. `-i off` specifies that counter instrumentation is enabled but initially turned off (refer to “mpirun” on page 49 and “MPI_INSTR” on page 41). Data collection begins after all processes collectively call `MPIHP_Trace_on`. HP MPI collects profiling information only for code between `MPIHP_Trace_on` and `MPIHP_Trace_off`

CAUTION **`MPIHP_Trace_on` and `MPIHP_Trace_off` are collective routines and must be called by all ranks in your application. Otherwise, the application deadlocks.**

Viewing ASCII instrumentation data

The ASCII instrumentation profile is a text file with the .instr extension. For example, to view the instrumentation file for the compute_pi.f application, you can print the *prefix*.instr file. If you defined *prefix* for the file as compute_pi, as you did when you created the instrumentation file in “Creating an instrumentation profile” on page 68, you would print compute_pi.instr.

The ASCII instrumentation profile provides the version, the date your application ran, and summarizes information according to application, rank, and routines. Figure 2 on page 71 is an example of an ASCII instrumentation profile.

The information available in the *prefix*.instr file includes:

- Overhead time—The time a process or routine spends inside MPI. For example, the time a process spends doing message packing.
- Blocking time—The time a process or routine is blocked waiting for a message to arrive before resuming execution.
- Communication hot spots—The processes in your application between which the largest amount of time is spent in communication.
- Message bin—The range of message sizes in bytes. The instrumentation profile reports the number of messages according to message length.

NOTE You do not get message size information for MPI_Alltoallv instrumentation.

Figure 2 displays the contents of the example report compute_pi.instr.

Figure 2 ASCII instrumentation profile

```
Version: HP MPI B6011/B6280 - HP-UX 10.20

Date:    Mon Feb  2 17:36:59 1998
Scale:   Wall Clock Seconds
Processes: 2
User:  33.65%
MPI:   66.35% [Overhead:66.35% Blocking:0.00%]

Total Message Count: 4
Minimum Message Range:            4 [0..32]
Maximum Message Range:            4 [0..32]
Average Message Range:            4 [0..32]

Top Routines:
MPI_Init        86.39% [Overhead:86.39% Blocking: 0.00%]
MPI_Bcast       12.96% [Overhead:12.96% Blocking: 0.00%]
MPI_Finalize     0.43% [Overhead: 0.43% Blocking: 0.00%]
MPI_Reduce       0.21% [Overhead: 0.21% Blocking: 0.00%]


------------------------------------------------------------
---------------  Instrumentation Data   ---------------
------------------------------------------------------------

Application Summary by Rank:
Rank    Duration     Overhead     Blocking   User    MPI
------------------------------------------------------------------
1       0.248998     0.221605     0.000000   11.00%  89.00%
0       0.249118     0.108919     0.000000   56.28%  43.72%
------------------------------------------------------------------

Routine Summary:

Routine         Calls   Overhead        Blocking
-----------------------------------------------------------
MPI_Init        2       0.285536        0.000000
min                     0.086926        0.000000
max                     0.198610        0.000000
avg                     0.142768        0.000000
MPI_Bcast       2       0.042849        0.000000
min                     0.021393        0.000000
max                     0.021456        0.000000
avg                     0.021424        0.000000
MPI_Finalize    2       0.001434        0.000000
min                     0.000240        0.000000
max                     0.001194        0.000000
avg                     0.000717        0.000000
MPI_Reduce      2       0.000705        0.000000
min                     0.000297        0.000000
max                     0.000408        0.000000
avg                     0.000353        0.000000
-----------------------------------------------------------
```

```
Routine Summary by Rank:
Routine         Rank    Calls   Overhead        Blocking
---------------------------------------------------------
MPI_Init        0       1       0.086926        0.000000
                1       1       0.198610        0.000000
MPI_Bcast       0       1       0.021456        0.000000
                1       1       0.021393        0.000000
MPI_Finalize    0       1       0.000240        0.000000
                1       1       0.001194        0.000000
MPI_Reduce      0       1       0.000297        0.000000
                1       1       0.000408        0.000000
---------------------------------------------------------

Routine Summary by Rank and Peer:
Routine         Rank    Peer    Calls   Overhead    Blocking
---------------------------------------------------------------
MPI_Bcast       0       0       1       0.021456    0.000000
                1       0       1       0.021393    0.000000
MPI_Reduce      0       0       1       0.000297    0.000000
                1       0       1       0.000408    0.000000
---------------------------------------------------------------

Message Summary:
Routine        Message Bin        Count
-----------------------------------------------------
MPI_Bcast       [0..32]           2
MPI_Reduce      [0..32]           2
-----------------------------------------------------

Message Summary by Rank:
Routine       Rank    Message Bin      Count
-----------------------------------------------------
MPI_Bcast     0       [0..32]          1
              1       [0..32]          1
MPI_Reduce    0       [0..32]          1
              1       [0..32]          1
-----------------------------------------------------

Message Summary by Rank and Peer:
Routine      Rank    Peer    Message Bin      Count
-----------------------------------------------------
MPI_Bcast    0       0       [0..32]          1
             1       0       [0..32]          1
MPI_Reduce   0       0       [0..32]          1
             1       0       [0..32]          1
-----------------------------------------------------------
```

Viewing instrumentation data with mpiview

The mpiview utility is a graphical user interface that displays instrumentation data collected at runtime by an MPI application. The following sections describe how to use mpiview to analyze your instrumentation data files:

- Loading an mpiview file
- Selecting a graph type
- Viewing multiple graphs
- Analyzing graphs

Loading an mpiview file

To view an instrumentation profile, invoke the mpiview utility and load your *prefix*.mpiview instrumentation file in one of the following ways:

- Provide the name of the instrumentation file when you invoke the mpiview utility. For example,

  % **`mpiview compute_pi.mpiview`**

  loads the compute_pi.mpiview file created in the mpirun example command above.

- Invoke mpiview without a filename. Enter

  % **`mpiview`**

  From the mpiview control window, select **File** from the menu bar, then **Open**. The mpiview utility displays a dialog box from which you can select your instrumentation file.

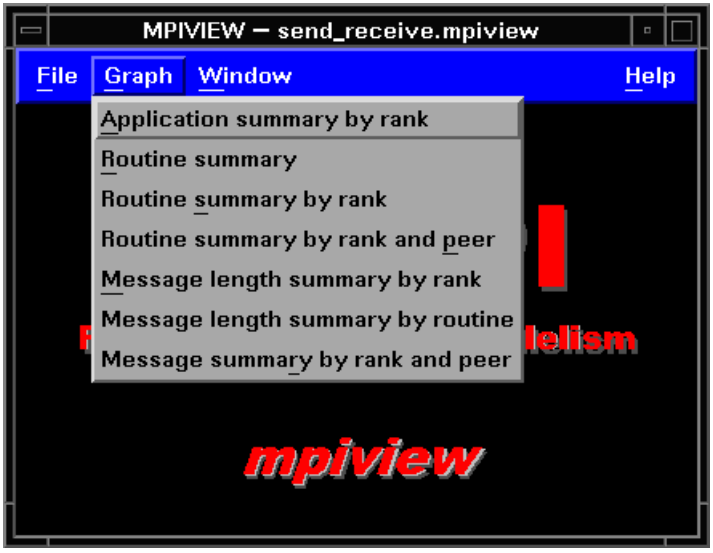
  After you select the file, mpiview displays a message stating either that the file was read successfully or that an error occurred.

Selecting a graph type

From the **Graph** pulldown menu on the main control window, select the type of graph you want to view. There are seven graph types that display your data in different formats. Each time you select a graph, mpiview displays it in a separate window.

Figure 3 displays the options on the **Graph** pulldown menu.

Figure 3 **MPIVIEW Graph menu**



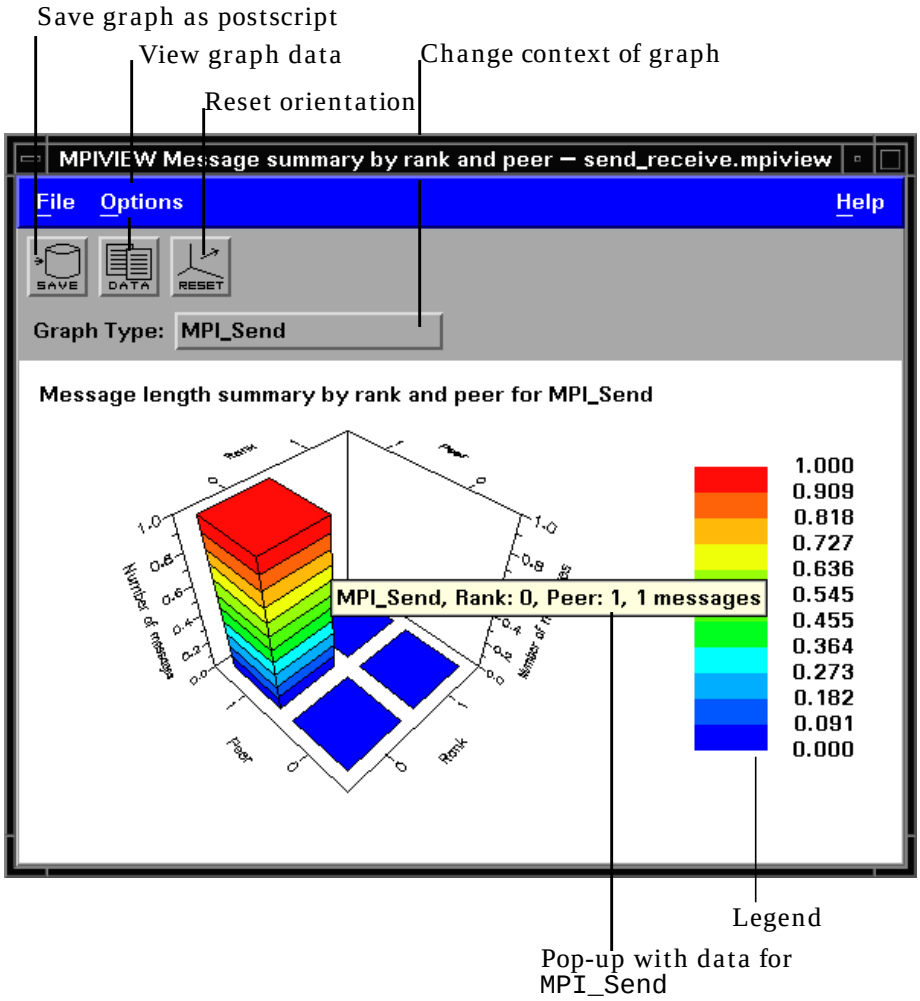
There are seven types from which to select:

- Application summary by rank—Displays data by rank.
- Routine summary—Displays data by routine.
- Routine summary by rank—Displays data by rank and routine.
- Routine summary by rank and peer—Displays data by rank and its peer rank for a given routine.
- Message length summary by rank—Displays data by routine and message length for a given rank or for all ranks.
- Message length summary by routine—Displays data by rank and message length for a given routine.
- Message length summary by rank and peer—Displays data by rank and its peer rank for a given routine.

Each time you select a graph, mpiview displays it in a separate window with the title of the graph and the filename of the data file used to generate it in the titlebar.

Figure 4 is an example of a graph window containing a “Message length summary by rank and peer” graph.

Figure 4 MPIVIEW graph window



Viewing multiple graphs

From the Window pulldown menu you can

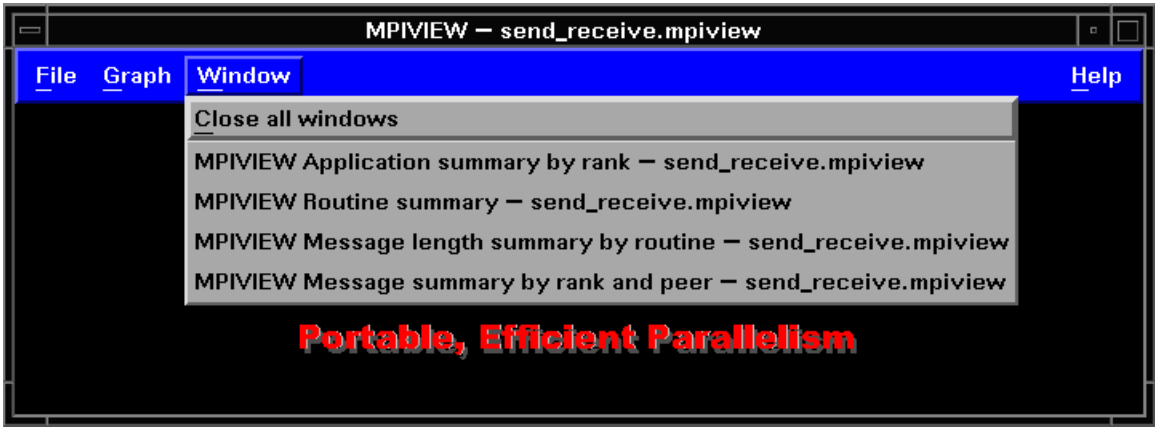
- Select one of the graphs from the list to view. The mpiview utility shuffles the window containing the selected graph to the top of your stack of overlapping windows.
- Select **Close all windows** to dismiss all the graphs from your display.

The mpiview utility does not impose a limit on the number of graphs it displays at any time. Each time you select a graph, mpiview displays it in a separate window with the title of the graph and the filename of the data file in the titlebar.

The mpiview **Window** pulldown menu initially contains only the **Close all windows** command. For each graph you invoke from the **Graph** pulldown menu, a new item appears in the **Window** pulldown menu. Each new item has the title of the graph, along with the name of the data file used to generate the graph.

Figure 5 displays an example of the **Window** menu containing the **Close all windows** option and four graph options.

Figure 5 MPIVIEW Window menu



Analyzing graphs

Each graph window provides functionality accessible through menu items, the toolbar, and using mouse manipulations. Table 7 describes the functionality available to help you analyze of your data.

Table 7 MPIVIEW analysis functions

Functionality	How to invoke
Save graph as a postscript file.	Select the **File** pulldown menu, then **Save as**, or select the **Save** icon on the toolbar.
Display graphed data in text format.	Select the **Options** pulldown menu, then **View Graph Data** or select the **Data** icon on the toolbar.
Reset a three dimensional graph to its original position after you rotate it or use the zoom feature.	Select the **Options** pulldown menu, then **Reset Orientation** or select the **Reset** icon on the toolbar.
Change the context of the graph.	Use the **Graph Type** radio button on the toolbar to select from a submenu of graph types.
View exact data values for regions.	Move the mouse over any bar in the graph and click the left mouse button. Data values display in a pop-up window beside the mouse arrow. For example, refer to the pop-up for `MPI_Send` in Figure 4 on page 75.
Rotate a three dimensional graph.	Place the cursor over the graph and hold down the middle mouse button while moving the mouse. You can restrict rotation to a single axis by pressing the **x**, **y**, or **z** key while moving the mouse.
Zoom on a particular section of a three dimensional graph.	Hold down the **Control** key and the left mouse button. Drag the mouse to stretch a rectangle over the area you want to zoom. Release the **Control** key and the mouse button.
Toggle the graph legend.	Select the **Options** pulldown menu, then **Show Legend**.

NOTE HP MPI 1.7 is the last release that will support mpiview.

mipview is not supported for Itanium-based systems.

Using XMPI

XMPI is an X/Motif graphical user interface for running applications, monitoring processes and messages, and viewing trace files. XMPI provides a graphical display of the state of processes within an HP MPI application. This functionality is supported for applications linked with the standard HP MPI library, but not for applications linked with the thread-compliant library or the diagnostic library.

XMPI is useful when analyzing programs at the application level (for example, examining HP MPI datatypes and communicators). You can run XMPI without having to recompile or relink your application.

XMPI runs in one of two modes: postmortem mode or interactive mode. In postmortem mode, you can view trace information for each process in your application. In interactive mode, you can monitor process communications by taking snapshots while your application is running.

The default X resource settings that determine how XMPI displays on your workstation are stored in /opt/mpi/lib/X11/app-defaults/XMPI. See Appendix B, “XMPI resource file” for a list of these settings.

Working with postmortem mode

To use XMPI's postmortem mode, you must first create a trace file. Load the trace file into XMPI to view state information for each process in your application.

Creating a trace file

To create a trace file, use the following syntax:

`mpirun -t` *spec* `-np #` *program*

as described in "mpirun" on page 49 and "Preparing XMPI files" on page 24.

By default, XMPI profiles the entire application from `MPI_Init` to `MPI_Finalize`. However, HP MPI provides nonstandard `MPIHP_Trace_on` and `MPIHP_Trace_off` routines to help troubleshoot application problems at finer granularity. To use `MPIHP_Trace_on` and `MPIHP_Trace_off`:

1. Insert the `MPIHP_Trace_on` and `MPIHP_Trace_off` pair around code that you want to profile.
2. Build the application and invoke `mpirun` with the `-t off` option.

   `-t off` specifies that tracing is enabled but initially turned off (refer to "mpirun" on page 49 and "MPI_XMPI" on page 47). Data collection begins after all processes collectively call `MPIHP_Trace_on`. XMPI collects trace information only for code between `MPIHP_Trace_on` and `MPIHP_Trace_off`.
3. Run the trace file in XMPI to identify problems during application execution.

CAUTION **`MPIHP_Trace_on` and `MPIHP_Trace_off` are collective routines and must be called by all ranks in your application. Otherwise, the application deadlocks.**

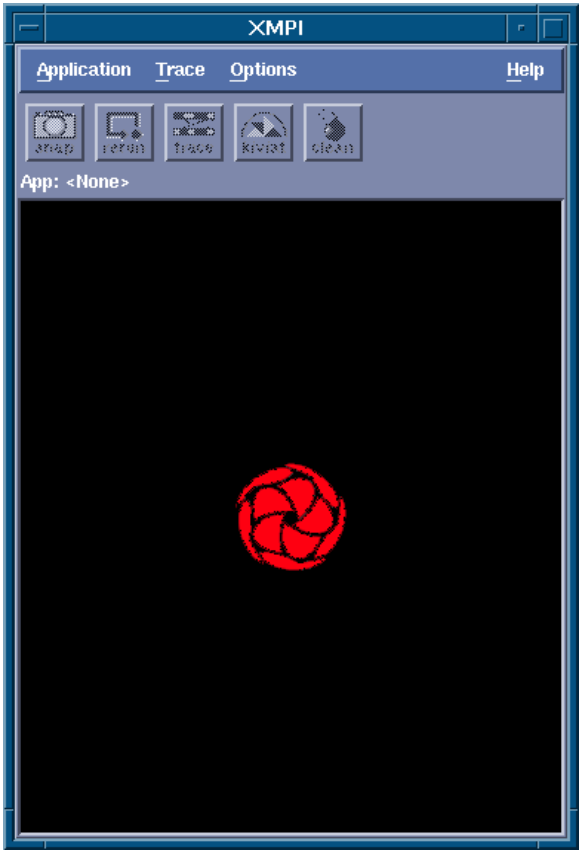
Viewing a trace file

Use the following instructions to view a trace file:

Step 1. Enter `xmpi` at your UNIX prompt to open the XMPI main window.

Refer to “xmpi” on page 61 for information about options you can specify with `xmpi`. Figure 6 shows the XMPI main window.

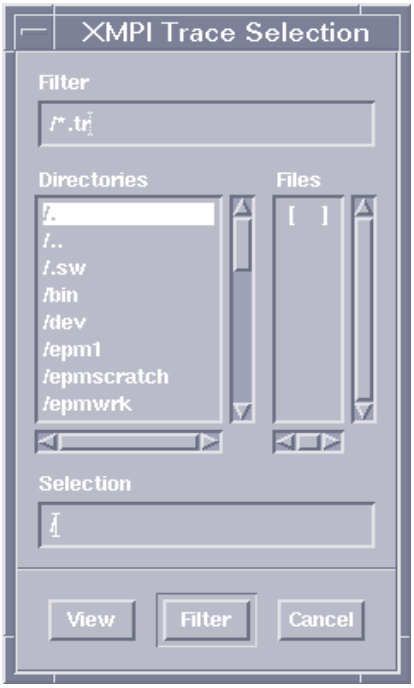
Figure 6 XMPI main window



Step 2. Select the **Trace** pull down menu on the main window, then **View**.

XMPI invokes the XMPI Trace Selection dialog in which you can find and select your trace file. Figure 7 shows the Trace Selection dialog.

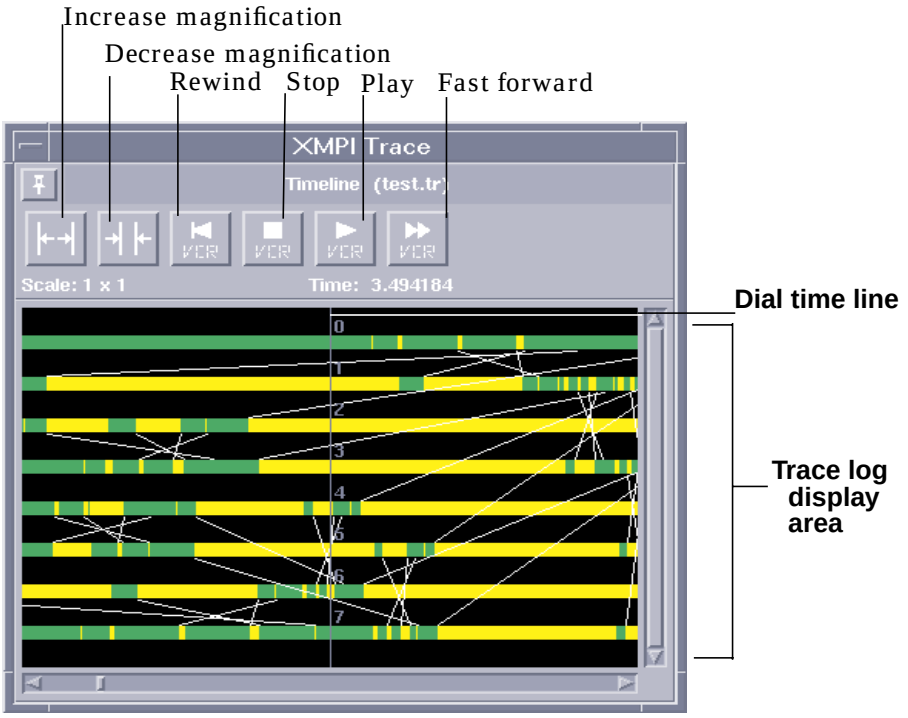
Figure 7 XMPI Trace Selection



Step 3. Select or type the full path name of the appropriate trace file in the Trace Selection dialog **Selection** field and select **View**.

XMPI invokes the XMPI Trace dialog. Figure 8 shows an example of a trace log.

Figure 8 XMPI trace log



When viewing trace files containing multiple segments, that is, multiple `MPIHP_Trace_on` and `MPIHP_Trace_off` pairs, XMPI prompts you for the number of the segment you want to view. To view different segments, reload the trace file and specify the new segment number when you get the prompt.

Figure 8 displays a typical XPMI Trace consisting of an icon bar, information about the current magnification and dial time, and a main window displaying the trace log.

The icon bar allows you to:

- Increase the magnification of the trace log.
- Decrease the magnification of the trace log.
- Rewind the trace log to the beginning—resets Dial time to the beginning.
- Stop playing the trace log.
- Play the trace log.
- Fast forward the trace log.

Refer to Figure 8 on page 82 to identify the icons and their functionality.

To set the magnification for viewing a trace file, select the **Increase** or **Decrease** icon on the icon bar. Dial time indicates how long the application has been running in seconds. The time is indicated on the toolbar.

The trace log display area shows a separate trace for each process in the application. Dial time is represented as a vertical line. The rank for each process is shown where the dial time line intersects a process trace.

Each process trace can have three colors:

Green	Represents the length of time a process runs outside of MPI.
Red	Represents the length of time a process is blocked, waiting for communication to finish before the process resumes execution.
Yellow	Represents a process's overhead time inside MPI (for example, time spent doing message packing).

Blocking point-to-point communications are represented by a trace for each process showing the time spent in system overhead and time spent blocked waiting for communication. A line between process traces connects the appropriate send and receive trace segments. The line starts at the beginning of the send segment and ends at the end of the receive segment.

For nonblocking point-to-point communications, XMPI draws a system overhead segment when a send and receive are initiated. When the communication is completed using a wait or a test, XMPI draws

segments showing system overhead and blocking time. Lines are drawn between matching sends and receives, except in this case, the line is drawn from the segment where the send was initiated to the segment where the corresponding receive completed.

Collective communications are represented by a trace for each process showing the time spent in system overhead and time spent blocked waiting for communication.

Some send and receive segments may not have a matching segment. In this case, a stub line is drawn out of the send segment or into the receive segment.

To play the trace file, select **Play** or **Fast Forward** on the icon bar. For any given dial time, the state of the processes is reflected in the main window and the Kiviat diagram as well as the trace log window. Refer to “Viewing process information” on page 85 and “Viewing Kiviat information” on page 89 to learn how to interpret the information.

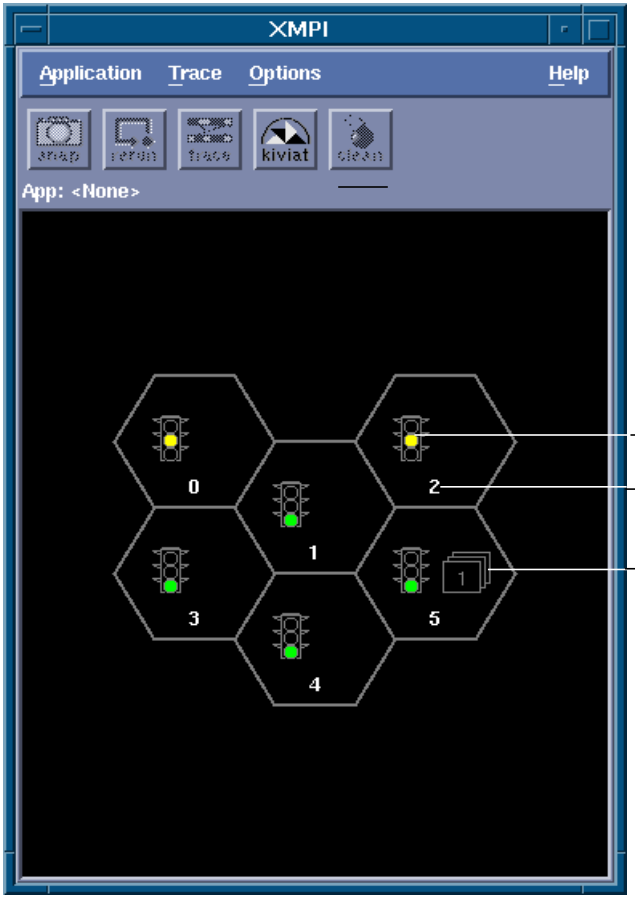
Viewing process information

When you play the trace file the state of the processes is reflected in the main window and the Kiviat diagram. The following instructions describe how to view process information in the main window:

Step 1. Start XMPI and open a trace for viewing as described in "Creating a trace file" on page 79.

The XMPI main window fills with a group of tiled hexagons, each representing the current state of a process and labelled by the process's rank within MPI_COMM_WORLD. Figure 9 shows the XMPI main window displaying hexagons representing six processes (ranks 0 through 5).

Figure 9 XMPI process information



The current state of a process is indicated by the color of the signal light in the hexagon.

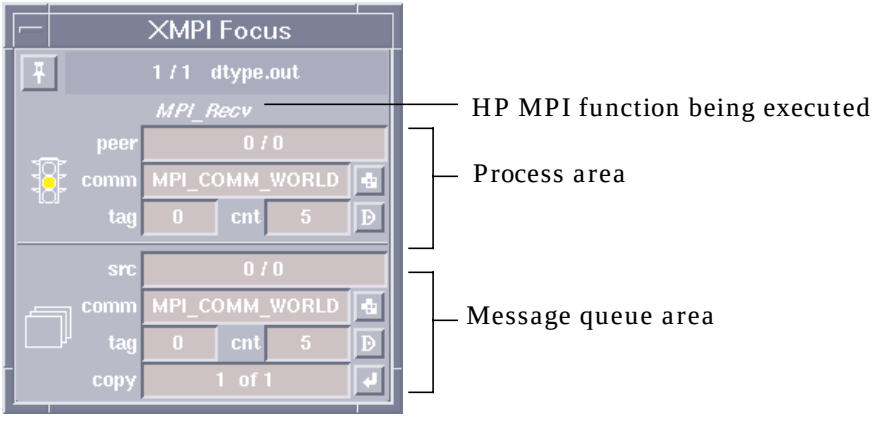
The color of the signal light corresponds to the color in the XMPI trace log for a given process. As the trace file plays and processes communicate with each other the signal light colors change.

Along with the signal light icon, hexagons may contain a second icon, indicating the number of messages sent to a process but not yet received.

Step 2. Click once on the hexagon representing the process for which you want more information.

XMPI displays the XMPI Focus dialog that has a process area and a message queue area. Figure 10 displays a Focus dialog.

Figure 10 XMPI Focus dialog



Values in the fields change as you play the trace file and processes communicate with each other.

The process area describes the state of a process together with the name and arguments for the HP MPI function being executed. The fields include:

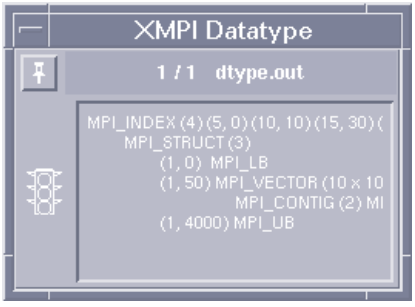
peer — Displays the rank of the function's peer process. A process is identified in the format *rank_x*/*rank_y* where *rank_x* indicates the rank of the process in MPI_COMM_WORLD, and *rank_y* indicates the rank of the process within the current communicator.

comm — Names the communicator used by the HP MPI function. When you select the icon to the right of the **comm** field, the hexagons for processes that belong to the communicator are highlighted in the XMPI main window.

tag — Displays the value of the tag argument associated with the message.

cnt — Shows the count of the message data elements associated with the message when it was sent. When you select the icon to the right of the **cnt** field XMPI opens the XMPI Datatype dialog as shown in Figure 11.

Figure 11 XMPI Datatype dialog



The XMPI Datatype dialog displays the type map of the datatype associated with the message when it was sent. The datatype can be one of the predefined datatypes or a user-defined datatype.

The datatype information changes as the trace file plays and processes communicate with each other.

The message queue area describes the current state of the queue of messages sent to the process but not yet received. The fields include:

src — Displays the rank of the process sending the message. A process is identified in the format *rank_x*/*rank_y* where *rank_x* indicates the rank of the process in MPI_COMM_WORLD, and *rank_y* indicates the rank of the process within the current communicator.

comm — Names the communicator used by the HP MPI function. When you select the icon to the right of the **comm** field the hexagons for processes that belong to the communicator are highlighted in the XMPI main window.

tag — Displays the value of the tag argument associated with the message when it was sent.

cnt — Shows the count of the message data elements associated with the message when it was sent. When you select the icon to the right of the **cnt** field XMPI opens the XMPI Datatype dialog. The XMPI Datatype dialog displays the type map of the datatype associated with the message when it was sent. Refer to Figure 11 on page 87 for the Datatype dialog.

copy — Displays the total number of messages and the number of messages of the type described in the current Focus dialog. The format is (*number_of_messages_of_the_type_described_in_the_current_focus_dialog*) of *(total_number_of_messages*).

A message type is defined by its message envelope consisting of the sender, the communicator, the tag, the count, and the datatype.

For example, if a process is waiting to receive 10 messages where six of the messages have one type of message envelope and the remaining four have another, the copy field toggles between 6 of 10 and 4 of 10. Use the icon to the right of the **copy** field to view the different Focus dialogs that exist to describe each message type.

XMPI treats six messages, each with the same envelope, as one copy and the remaining four messages as a different copy. This way, one Focus dialog is necessary for each message type and not for each individual message. For example, if a communication involves a hundred messages all having the same envelope, you can work with a single Focus dialog not with one hundred copies.

Step 3. Select the **Application** menu, then **Quit** to close XMPI.

Viewing Kiviat information

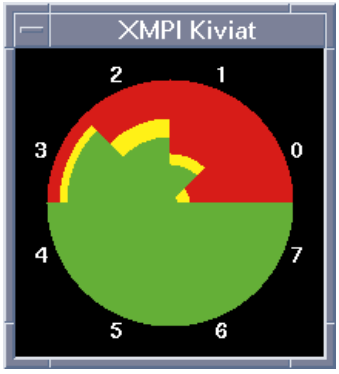
When you play the trace file the state of the processes is reflected in the main window and the Kiviat graph. Use the following instructions to view performance information in a Kiviat graph:

Step 1. Start XMPI and open a trace for viewing as described in "Creating a trace file" on page 79.

Step 2. Select **Kiviat** from the **Trace** menu.

XMPI opens a window containing a Kiviat graph as shown in Figure 12.

Figure 12 XMPI Kiviat



Red: Process blocked

Yellow: MPI overhead

Green: Process running outside MPI

The XMPI Kiviat shows, in segmented pie-chart format, the cumulative time up to the current dial time spent by each process in running, overhead, and blocked states represented by green, yellow, and red respectively. The process numbers are indicated on the graph.

As the trace file plays and processes communicate, the Kiviat changes to reflect the time spent running, blocked, or in MPI overhead.

Use the XMPI Kiviat to determine whether processes are load balanced and applications are synchronized. If an application is load balanced, the amount of time processes spend in each state should be equal. If an application is synchronized, the segments representing each of the three states should be concentric.

Step 3. Select the **Application** menu, then **Quit** to close XMPI.

Working with interactive mode

Interactive mode allows you to load and run an appfile to view state information for each process as your application runs.

Running an appfile

Use these instructions to run and view your appfile:

Step 1. Enter `xmpi` at your UNIX prompt to open the XMPI main window.

Refer to “xmpi” on page 61 for information about options you can specify with `xmpi`. Figure 6 on page 80 shows the XMPI main window.

Step 2. Select the **Application** menu, then **Browse&Run**.

XMPI opens the XMPI Application Browser dialog.

Step 3. Select or type the full path name of the appropriate appfile in the **Selection** field and select **Run**.

The XMPI main window fills with a group of tiled hexagons, each representing the current state of a process and labelled by the process’s rank within MPI_COMM_WORLD.

The window is the same as the one XMPI invokes in postmortem mode. Refer to Figure 9 on page 85. The state of a process is indicated by the color of the signal light in the hexagon. Along with the signal light icon, hexagons can contain an icon that indicates the number of messages sent to a process that it has yet to receive.

The process hexagons persist only as long as the application runs and disappear when the application completes.

To monitor and analyze your application when running interactive mode XMPI provides the following functionality:

- Snapshot utility

  The snapshot utility helps you debug applications that hang. If automatic snapshot is enabled, XMPI takes periodic snapshots of the application and displays state information for each process in the XMPI main window, the XMPI Focus dialog, and the XMPI Datatype dialog. You can use this information to view the state of each process when an application hangs.

  Refer to "Changing default settings and viewing options" on page 95 for information to enable automatic snapshot. Refer to Figure 10 on page 86, and Figure 11 on page 87 for details about the XMPI Focus and Datatype dialogs.

  If automatic snapshot is disabled, XMPI displays information for each process when the application begins, but does not update the information as the application runs.

  You can take application snapshots manually by selecting the **Application** pulldown menu, then **Snapshot**. XMPI displays information for each process, but this information is not updated until you take the next snapshot. You can only take snapshots when an appfile is running and you cannot replay snapshots like trace files.

- Dump utility

  The Dump utility consolidates all trace file data collected up to the current time in the application's life-span into a single output file, *prefix*.tr. Define *prefix*, in the XMPI Dump dialog, as the name you want to give your .tr file. Refer to Figure 13 on page 92 for the XMPI Dump dialog.

  The Dump utility is only available if you first enable runtime trace generation for all application processes as follows:

  – Select **Options** from the main window's pulldown menu, then **mpirun**. XMPI invokes an mpirun options dialog.

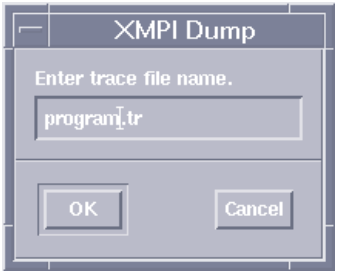
  – Select **Tracing** in the mpirun options dialog.

  – Enter a prefix for the .tr file in the **Prefix** field.

Refer to “Changing default settings and viewing options” on page 95 for more details about enabling runtime trace generation and the mpirun options dialog.

At any time while your application is running, you can select **Dump** from the **Trace** menu. XMPI invokes the Dump dialog, displayed in Figure 13.

Figure 13 XMPI Dump dialog



Specify the name of the consolidated .tr output file. The name you specified in the **Prefix** field in the mpirun options trace dialog is entered by default. You can use this name or type another.

After you have created the .tr output file, you can resume snapshot monitoring.

- Express utility

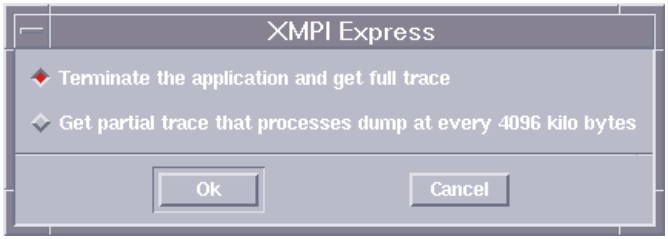
  The Express utility allows generation of an XMPI Trace log using the data collected up to the current time in the application's life-span. Refer to Figure 8 on page 82 for an example of a Trace log.

  Express, like the Dump utility, is only available if you first enable runtime trace generation for all application processes by selecting the **Options** pulldown menu, then **mpirun**, and then the **Tracing** button on the mpirun options trace dialog.

  To invoke the XMPI Express dialog, select the **Trace** pulldown menu, then **Express**, while your application is running.

Figure 14 displays the XMPI Express dialog.

Figure 14 XMPI Express dialog


XMPI Express
Terminate the application and get full trace
Get partial trace that processes dump at every 4096 kilo bytes
Ok
Cancel

Select one of two options from the dialog:

- **Terminate the application and get full trace**

  Specifies that the content of each process buffer is written to a trace file. The write happens whether process buffers are partially or totally full. The trace files for each process are consolidated in a *prefix*.tr output file, where *prefix* is the name you specified in the **Prefix** field of the Tracing options dialog (see Figure 18 on page 98). XMPI loads and displays the *prefix*.tr output file in the XMPI Trace window.

  When you select this field, XMPI displays the XMPI Confirmation dialog to confirm that you want to terminate the application. You must select **Yes** before processing will continue.

  After XMPI loads and displays the .tr output file in the XMPI Trace window, you cannot resume snapshot monitoring because the application has terminated.

- **Get partial trace that processes dump at every 4096 kilobytes**

  Specifies that the content of each process buffer is written to a trace file only after the buffer becomes full. The trace files are then consolidated to a *prefix*.tr output file, where *prefix* is the name you specified in the **Prefix** field of the Tracing options dialog (see Figure 18 on page 98). XMPI loads and displays the *prefix*.tr output file in the XMPI Trace window.

  After XMPI loads and displays the .tr output file in the XMPI Trace window, you cannot resume snapshot monitoring even though the application may still be running.

In interactive mode, XMPI gathers and displays data from the running appfile or a trace file.

When an application is running, the data source is the appfile, and automatic snapshot is enabled. Even though the application may be creating trace data, the snapshot function does not use it. Instead, the snapshot function acquires data from internal hooks in HP MPI.

At any point in interactive mode, you can load and view a trace file by selecting the **Trace** menu, then the **View** or **Express** command.

When you use the **View** or **Express** command to load and view a trace file, the data source switches to the loaded trace file, and the snapshot function is disabled. You must rerun your application to switch the data source from a trace file back to an appfile.

Step 4. Select **Clean** from the **Application** menu at any time in interactive mode to kill the application and close any associated XMPI Focus and XMPI Datatype dialogs.

XMPI displays the XMPI Confirmation dialog to confirm that you want to terminate the application.

Step 5. Select **Yes** to terminate your application and close associated dialogs. You can run another application by selecting an appfile from the XMPI Application Browser dialog.

Changing default settings and viewing options

You should initially run your appfile using the XMPI default settings. You can change XMPI default settings and profile viewing options from the **Options** pulldown menu. The **Options** menu has three commands:

Monitoring...	Controls automatic snapshot
Buffers...	Controls buffer size for processes
mpirun...	Controls tracing options

Use the following instructions to change the XMPI default settings and your viewing options:

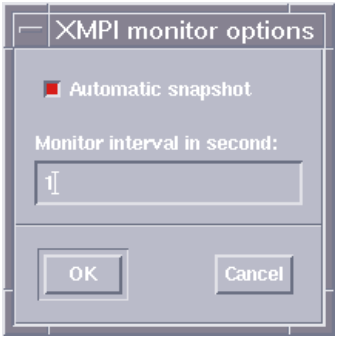
Step 1. Enter `xmpi` to open the XMPI main window.

You can specify options to change the default XMPI window settings (size, color, position, etc.). Refer to “xmpi” on page 61 for details.

Step 2. Select the **Options** menu, then **Monitoring**.

XMPI opens the XMPI monitor options dialog as shown in Figure 15.

Figure 15 XMPI monitor options dialog



The fields include:

- **Automatic snapshot**

  Enables the automatic snapshot function. If automatic snapshot is enabled, XMPI takes snapshots of the application you are running and displays state information for each process.

If automatic snapshot is disabled, XMPI displays information for each process when the application begins. However, you can only update this information manually. Disabling automatic snapshot may lead to buffer overflow problems because the contents of each process buffer are unloaded every time a snapshot is taken. For communication intensive applications, process buffers can quickly fill and overflow.

You can enable or disable automatic snapshot while your application is running. This can be useful during troubleshooting when the application runs to a certain point and you want to disable automatic snapshot to study process state information.

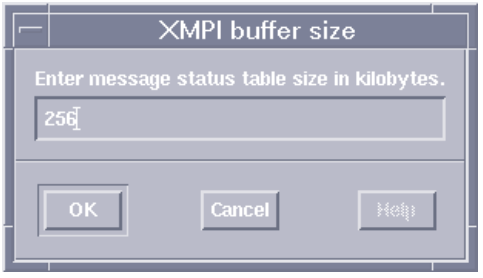
- **Monitor interval in seconds**

  Specifies, in seconds, how often XMPI takes a snapshot when automatic snapshot is enabled.

Step 3. Select **Buffers** from the **Options** menu.

XMPI opens the XMPI buffer size dialog as shown in Figure 16.

Figure 16 XMPI buffer size dialog

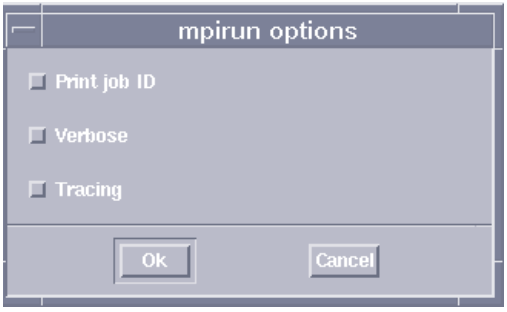


Specify the size, in kilobytes, for each process buffer. When you run an application, state information for each process is stored in a separate buffer. You may need to increase buffer size if overflow problems occur.

Step 4. Select **mpirun** from the **Options** menu.

XMPI opens the mpirun options dialog as shown in Figure 17.

Figure 17 mpirun options dialog

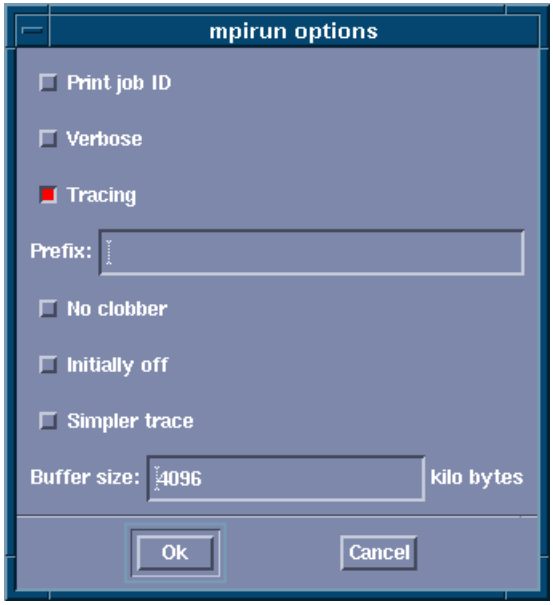


The fields include:

Print job ID	Enables printing of the HP MPI job ID.
Verbose	Enables verbose mode.
Tracing	Enables runtime trace generation for all application processes. When you select **Tracing**, XMPI expands the options dialog to include more tracing options, as shown in Figure 18.

Figure 18 displays the expanded Tracing options dialog.

Figure 18 Tracing options dialog



The fields you can use to specify tracing options are:

Prefix	Specifies the prefix name for the file where process write trace data. The trace files for each process are consolidated to a *prefix*.tr output file. This is a required field.
No clobber	Specifies no clobber, which means that an HP MPI application aborts if a file with the name specified in the **Prefix** field already exists.
Initially off	Specifies that trace generation is initially turned off.
Simpler trace	Specifies a simpler tracing mode by omitting `MPI_Test`, `MPI_Testall`, `MPI_Testany`, and `MPI_Testsome` calls that do not complete a request.

Buffer size	Denotes the buffering size in kilobytes for dumping process trace data. Actual buffering size may be rounded up by the system. The default buffering size is 4096 kilobytes. Specifying a large buffering size reduces the need to flush trace data to a file when process buffers reach capacity. Flushing frequently can increase the overhead for I/O.

NOTE HP MPI 1.7 is the last release that will support XMPI.

XMPI is not supported for Itanium-based systems.

Using CXperf

CXperf allows you to profile each process in an HP MPI application. Profile information is stored in a separate performance data file (PDF) for each process. To analyze your profiling data using CXperf, you must first use the `merge` utility to merge the data from the separate files into a single PDF. Refer to the merge(1) man page.

Using CXperf you can instrument your application to collect performance using one or more of the following metrics:

- Wall clock time
- CPU time
- Execution counts
- Cache misses
- Latency
- Migrations
- Context switches
- Page faults
- Instruction counts
- Data translation lookaside buffer (DTLB) misses
- Instruction translation lookaside buffer (ITLB) misses

You can display the data as a 3D (Parallel) profile, a 2D (Summary) profile, a text report, or a dynamic call graph. For more information, refer to the *CXperf User's Guide* and the *CXperf Command Reference*.

Using the profiling interface

The MPI profiling interface provides a mechanism by which implementors of profiling tools can collect performance information without access to the underlying MPI implementation source code.

Because HP MPI provides several options for profiling your applications, you may not need the profiling interface to write your own routines. HP MPI makes use of MPI profiling interface mechanisms to provide the diagnostic library for debugging. In addition, HP MPI provides tracing and lightweight counter instrumentation. For details, refer to

- “Using counter instrumentation” on page 68
- “Using XMPI” on page 78
- “Using the diagnostics library” on page 118

The profiling interface allows you to intercept calls made by the user program to the MPI library. For example, you may want to measure the time spent in each call to a certain library routine or create a log file. You can collect your information of interest and then call the underlying MPI implementation through a name shifted entry point.

All routines in the HP MPI library begin with the `MPI_` prefix. Consistent with the “Profiling Interface” section of the MPI 1.2 standard, routines are also accessible using the `PMPI_` prefix (for example, `MPI_Send` and `PMPI_Send` access the same routine).

To use the profiling interface, write wrapper versions of the MPI library routines you want the linker to intercept. These wrapper routines collect data for some statistic or perform some other action. The wrapper then calls the MPI library routine using its `PMPI_` prefix.

Fortran profiling interface

To facilitate improved Fortran performance, we no longer implement Fortran calls as wrappers to C calls. Consequently, profiling routines built for C calls will no longer cause the corresponding Fortran calls to be wrapped automatically. In order to profile Fortran routines, separate wrappers need to be written for the Fortran calls.

For example:

```
#include <stdio.h>
#include <mpi.h>

int MPI_Send(void *buf, int count, MPI_Datatype type,
             int to, int tag, MPI_Comm comm)
{
     printf("Calling C MPI_Send to %d\n", to);
     return PMPI_Send(buf, count, type, to, tag, comm);

}

#pragma _HP_SECONDARY_DEF mpi_send mpi_send_
void mpi_send(void *buf, int *count, int *type, int *to,
             int *tag, int *comm, int *ierr)

{

     printf("Calling Fortran MPI_Send to %d\n", *to);
     pmpi_send(buf, count, type, to, tag, comm, ierr);

}
```

5 Tuning

This chapter provides information about tuning HP MPI applications to improve performance. The topics covered are:

- MPI_FLAGS options
- Message latency and bandwidth
- Multiple network interfaces
- Processor subscription
- MPI routine selection
- Multilevel parallelism
- Coding considerations

The tuning information in this chapter improves application performance in most but not all cases. Use this information together with the output from counter instrumentation, mpiview, or XMPI to determine which tuning changes are appropriate to improve your application's performance.

When you develop HP MPI applications, several factors can affect performance, whether your application runs on a single computer or in an environment consisting of multiple computers in a network. These factors are outlined in this chapter.

MPI_FLAGS options

By default, HP MPI validates all function parameters for all MPI function calls. If you have a well-behaved application, you can turn off argument checking by setting `MPI_FLAGS=E` to improve performance.

If you are running an application stand-alone on a dedicated system, setting `MPI_FLAGS=y` allows MPI to busy spin, thereby improving latency. See “MPI_FLAGS” on page 37 for more information on the y option.

Message latency and bandwidth

Latency is the time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

Latency is often dependent upon the length of messages being sent. An application's messaging behavior can vary greatly based upon whether a large number of small messages or a few large messages are sent.

Message bandwidth is the reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in megabytes per second. Bandwidth becomes important when message sizes are large.

To improve latency or bandwidth or both:

- Reduce the number of process communications by designing coarse-grained applications.
- Use derived, contiguous data types for dense data structures to eliminate unnecessary byte-copy operations in certain cases. Use derived data types instead of `MPI_Pack` and `MPI_Unpack` if possible. HP MPI optimizes noncontiguous transfers of derived data types.
- Use collective operations whenever possible. This eliminates the overhead of using `MPI_Send` and `MPI_Recv` each time when one process communicates with others. Also, use the HP MPI collectives rather than customizing your own.
- Specify the source process rank whenever possible when calling MPI routines. Using `MPI_ANY_SOURCE` may increase latency.
- Double-word align data buffers if possible. This improves byte-copy performance between sending and receiving processes because of double-word loads and stores.
- Use `MPI_Recv_init and MPI_Startall` instead of a loop of `MPI_Irecv` calls in cases where requests may not complete immediately.

For example, suppose you write an application with the following code section:

```
j = 0
for (i=0; i<size; i++) {
   if (i==rank) continue;
   MPI_Irecv(buf[i], count, dtype, i, 0, comm, &requests[j++]);
}
MPI_Waitall(size-1, requests, statuses);
```

Suppose that one of the iterations through `MPI_Irecv` does not complete before the next iteration of the loop. In this case, HP MPI tries to progress both requests. This progression effort could continue to grow if succeeding iterations also do not complete immediately, resulting in a higher latency.

However, you could rewrite the code section as follows:

```
j = 0
for (i=0; i<size; i++) {
   if (i==rank) continue;
   MPI_Recv_init(buf[i], count, dtype, i, 0, comm,
                                      &requests[j++]);
}
MPI_Startall(size-1, requests);
MPI_Waitall(size-1, requests, statuses);
```

In this case, all iterations through `MPI_Recv_init` are progressed just once when `MPI_Startall` is called. This approach avoids the additional progression overhead when using `MPI_Irecv` and can reduce application latency.

Multiple network interfaces

You can use multiple network interfaces for interhost communication while still having intrahost exchanges. In this case, the intrahost exchanges use shared memory between processes mapped to different same-host IP addresses.

To use multiple network interfaces, you must specify which MPI processes are associated with each IP address in your appfile.

For example, when you have two hosts, host0 and host1, each communicating using two ethernet cards, ethernet0 and ethernet1, you have four host names as follows:

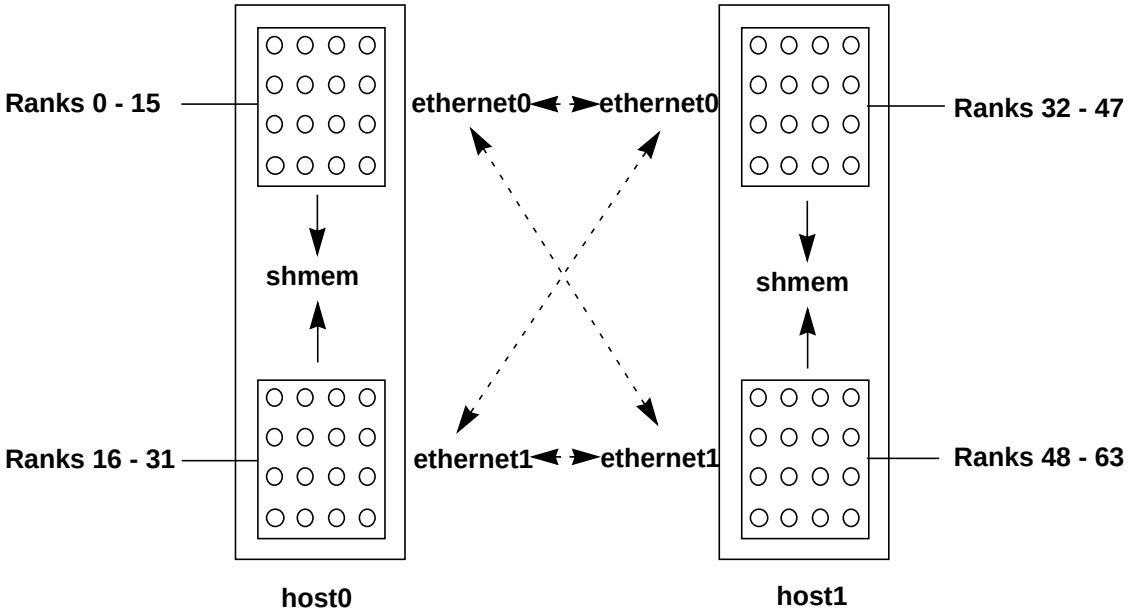
- host0-ethernet0
- host0-ethernet1
- host1-ethernet0
- host1-ethernet1

If your executable is called beavis.exe and uses 64 processes, your appfile should contain the following entries:

```
-h host0-ethernet0 -np 16 beavis.exe
-h host0-ethernet1 -np 16 beavis.exe
-h host1-ethernet0 -np 16 beavis.exe
-h host1-ethernet1 -np 16 beavis.exe
```

Now, when the appfile is run, 32 processes run on host0 and 32 processes run on host1 as shown in Figure 19.

Figure 19 Multiple network interfaces



Host0 processes with rank 0 - 15 communicate with processes with rank 16 - 31 through shared memory (shmem). Host0 processes also communicate through the host0-ethernet0 and the host0-ethernet1 network interfaces with host1 processes.

Processor subscription

Subscription refers to the match of processors and active processes on a host. Table 8 lists possible subscription types.

Table 8 Subscription types

Subscription type	Description
Under subscribed	More processors than active processes
Fully subscribed	Equal number of processors and active processes
Over subscribed	More active processes than processors

When a host is over subscribed, application performance decreases because of increased context switching.

Context switching can degrade application performance by slowing the computation phase, increasing message latency, and lowering message bandwidth. Simulations that use timing–sensitive algorithms can produce unexpected or erroneous results when run on an over-subscribed system.

In a situation where your system is oversubscribed but your MPI application is not, you can use gang scheduling to improve performance. Refer to "Gang scheduling" for details.

MPI routine selection

To achieve the lowest message latencies and highest message bandwidths for point-to-point synchronous communications, use the MPI blocking routines `MPI_Send` and `MPI_Recv`. For asynchronous communications, use the MPI nonblocking routines `MPI_Isend` and `MPI_Irecv`.

When using blocking routines, try to avoid pending requests. MPI must advance nonblocking messages, so calls to blocking receives must advance pending requests, occasionally resulting in lower application performance.

For tasks that require collective operations, use the appropriate MPI collective routine. HP MPI takes advantage of shared memory to perform efficient data movement and maximize your application's communication performance.

Multilevel parallelism

There are several ways to improve the performance of applications that use multilevel parallelism:

- Use the MPI library to provide coarse-grained parallelism and a parallelizing compiler to provide fine-grained (that is, thread-based) parallelism. An appropriate mix of coarse- and fine-grained parallelism provides better overall performance.
- Assign only one multithreaded process per host when placing application processes. This ensures that enough processors are available as different process threads become active.

Coding considerations

The following are suggestions and items to consider when coding your MPI applications to improve performance:

- Use HP MPI collective routines instead of coding your own with point-to-point routines because HP MPI's collective routines are optimized to use shared memory where possible for performance.

  Use commutative MPI reduction operations.

  – Use the MPI predefined reduction operations whenever possible because they are optimized.

  – When defining your own reduction operations, make them commutative. Commutative operations give MPI more options when ordering operations allowing it to select an order that leads to best performance.

- Use MPI derived datatypes when you exchange several small size messages that have no dependencies.

- Minimize your use of MPI_Test() polling schemes to minimize polling overhead.

- Code your applications to avoid unnecessary synchronization. In particular, strive to avoid MPI_Barrier calls. Typically an application can be modified to achieve the same end result using targeted synchronization instead of collective calls. For example, in many cases a token-passing ring may be used to achieve the same coordination as a loop of barrier calls.

6 Debugging and troubleshooting

This chapter describes debugging and troubleshooting HP MPI applications. The topics covered are:

- Debugging HP MPI applications
  - Using a single-process debugger
  - Using a multi-process debugger
  - Using the diagnostics library
  - Enhanced debugging output
  - Backtrace functionality
- Troubleshooting HP MPI applications
  - Building
  - Starting
  - Running
  - Completing
- Frequently asked questions

Debugging HP MPI applications

HP MPI allows you to use single-process debuggers to debug applications. The available debuggers are ADB, DDE, XDB, WDB and GDB. You access these debuggers by setting options in the `MPI_FLAGS` environment variable. HP MPI also supports the multithread, multiprocess debugger, TotalView on HP-UX 11.0 and later.

In addition to the use of debuggers, HP MPI provides a diagnostic library (DLIB) for advanced error checking and debugging. Another useful debugging tool, especially for deadlock investigations, is the XMPI utility. HP MPI also provides options to the environment variable `MPI_FLAGS` that report memory leaks (`l`), force MPI errors to be fatal (`f`), print the MPI job ID (`j`), and other functionality.

This section discusses single- and multi-process debuggers and the diagnostic library; refer to “MPI_FLAGS” on page 37 and “Using XMPI” on page 78 for information about using MPI_FLAGS option and XMPI, respectively.

Using a single-process debugger

Because HP MPI creates multiple processes and ADB, DDE, XDB, WDB, and GDB only handle single processes, HP MPI starts one debugger session per process. HP MPI creates processes in `MPI_Init`, and each process instantiates a debugger session. Each debugger session in turn attaches to the process that created it. HP MPI provides `MPI_DEBUG_CONT` to avoid a possible race condition while the debugger session starts and attaches to a process. `MPI_DEBUG_CONT` is an environment variable that HP MPI uses to temporarily halt debugger progress beyond MPI_Init. By default, `MPI_DEBUG_CONT` is set to 0 and you must reset it to 1 to allow the debug session to continue past `MPI_Init`.

The following procedure outlines the steps to follow when you use a single-process debugger:

Step 1. Set the `eadb`, `exdb`, `edde`, `ewdb` or `egdb` option in the `MPI_FLAGS` environment variable to use the ADB, XDB, DDE, WDB, or GDB debugger respectively. Refer to “MPI_FLAGS” on page 37 for information about MPI_FLAGS options.

Step 2. On remote hosts, set DISPLAY to point to your console. In addition, use `xhost` to allow remote hosts to redirect their windows to your console.

Step 3. Run your application.

When your application enters `MPI_Init`, HP MPI starts one debugger session per process and each debugger session attaches to its process.

Step 4. Set a breakpoint anywhere following `MPI_Init` in each session.

Step 5. Set the global variable `MPI_DEBUG_CONT` to 1 using each session's command line interface or graphical user interface. The syntax for setting the global variable depends upon which debugger you use:

(adb) **`mpi_debug_cont/w 1`**

(dde) **`set mpi_debug_cont = 1`**

(xdb) **`print *MPI_DEBUG_CONT = 1`**

(wdb) **`set MPI_DEBUG_CONT = 1`**

(gdb) **`set MPI_DEBUG_CONT = 1`**

Step 6. Issue the appropriate debugger command in each session to continue program execution.

Each process runs and stops at the breakpoint you set after `MPI_Init`.

Step 7. Continue to debug each process using the appropriate commands for your debugger.

Using a multi-process debugger

HP MPI supports the TotalView debugger on HP-UX version 11.0 and later. The preferred method when you run TotalView with HP MPI applications is to use the mpirun runtime utility command.

For example,

```
% mpicc myprogram.c -g
% mpirun -tv -np 2 a.out
```

In this example, myprogram.c is compiled using the HP MPI compiler utility for C programs (refer to "Compiling and running your first application" on page 19). The executable file is compiled with source line information and then mpirun runs the a.out MPI program:

`-g`	Specifies that the compiler generate the additional information needed by the symbolic debugger.
`-np` *2*	Specifies the number of processes to run (2, in this case).
`-tv`	Specifies that the MPI ranks are run under TotalView.

Alternatively, use mpirun to invoke an appfile:

```
% mpirun -tv -f my_appfile
```

`-tv`	Specifies that the MPI ranks are run under TotalView.
`-f` *appfile*	Specifies that mpirun parses *my_appfile* to get program and process count information for the run. Refer to "Creating an appfile" on page 55 for details about setting up your appfile.

Refer to "mpirun" on page 49 for details about mpirun.

Refer to the "MPI_FLAGS" on page 37 and the TotalView documentation for details about `MPI_FLAGS` and TotalView command line options, respectively.

By default, mpirun searches for TotalView in your `PATH` settings.You can also define the absolute path to TotalView using the `TOTALVIEW` environment variable:

```
% setenv TOTALVIEW /opt/totalview/bin/totalview [totalview-options]
```

The `TOTALVIEW` environment variable is used by mpirun.

NOTE When attaching to a running MPI application, you should attach to the MPI daemon process to enable debugging of all the MPI ranks in the application. You can identify the daemon process as the one at the top of a hierarchy of MPI jobs (the daemon also usually has the lowest PID among the MPI jobs).

Limitations

The following limitations apply to using TotalView with HP MPI applications:

1. All the executable files in your multihost MPI application must reside on your local machine, that is, the machine on which you start TotalView. Refer to "TotalView multihost example" on page 117 for details about requirements for directory structure and file locations.
2. TotalView sometimes displays extra HP-UX threads that have no useful debugging information. These are kernel threads that are created to deal with page and protection faults associated with one-copy operations that HP MPI uses to improve performance. You can ignore these kernel threads during your debugging session.

TotalView multihost example

The following example demonstrates how to debug a typical HP MPI multihost application using TotalView, including requirements for directory structure and file locations.

The MPI application is represented by an appfile, named my_appfile, which contains the following two lines:

```
-h local_host -np 2 /path/to/program1
-h remote_host -np 2 /path/to/program2
```

my_appfile resides on the local machine (local_host) in the /work/mpiapps/total directory.

To debug this application using TotalView (in this example, TotalView is invoked from the local machine):

1. Place your binary files in accessible locations.

- /path/to/program1 exists on local_host
- /path/to/program2 exists on remote_host

To run the application under TotalView, the directory layout on your local machine, with regard to the MPI executable files, must mirror the directory layout on each remote machine. Therefore, in this case, your setup must meet the following additional requirement:

- /path/to/program2 exists on local_host

2. In the /work/mpiapps/total directory on local_host, invoke TotalView by passing the `-tv` option to mpirun:

```
% mpirun -tv -f my_appfile
```

Using the diagnostics library

HP MPI provides a diagnostics library (DLIB) for advanced run time error checking and analysis. DLIB provides the following checks:

- Message signature analysis—Detects type mismatches in MPI calls. For example, in the two calls below, the send operation sends an integer, but the matching receive operation receives a floating-point number.

```
if (rank == 1) then
  MPI_Send(&buf1, 1, MPI_INT, 2, 17, MPI_COMM_WORLD);
else if (rank == 2)
  MPI_Recv(&buf2, 1, MPI_FLOAT, 1, 17, MPI_COMM_WORLD,
                                                &status);
```

- MPI object-space corruption—Detects attempts to write into objects such as MPI_Comm, MPI_Datatype, MPI_Request, MPI_Group, and MPI_Errhandler.

- Multiple buffer writes—Detects whether the data type specified in a receive or gather operation causes MPI to write to a user buffer more than once.

To disable these checks or enable formatted or unformatted printing of message data to a file, set the `MPI_DLIB_FLAGS` environment variable options appropriately. See “MPI_DLIB_FLAGS” on page 35 for more information.

To use the diagnostics library, specify the `-ldmpi` option when you compile your application.

NOTE Using DLIB reduces application performance. DLIB is not thread-compliant. Also, you cannot use DLIB with instrumentation or XMPI tracing.

Enhanced debugging output

HP MPI 1.7 provides improved readability and usefulness of MPI processes stdout and stderr. More intuitive options have been added for handling standard input:

- `Directed`: Input is directed to a specific MPI process.
- `Broadcast`: Input is copied to the stdin of all processes.
- `Ignore`: Input is ignored.

The default behavior is standard input is ignored.

Additional options are available to avoid confusing interleaving of output:

- Line buffering, block buffering, or no buffering
- Prepending of processes ranks to their stdout and stderr
- Simplification of redundant output

Backtrace functionality

HP MPI 1.7 handles several common termination signals differently than earlier versions of HP MPI. If any of the following signals are generated by an MPI application, a stack trace is printed prior to termination:

- `SIGBUS` - bus error
- `SIGSEGV` - segmentation violation
- `SIGILL` - illegal instruction
- `SIGSYS` - illegal argument to system call

The backtrace is helpful in determining where the signal was generated and the call stack at the time of the error. If a signal handler is established by the user code before calling `MPI_Init`, no backtrace will be printed for that signal type and the user's handler will be solely responsible for handling the signal. Any signal handler installed after `MPI_Init` will also override the backtrace functionality for that signal after the point it is established. If multiple processes cause a signal, each of them will print a backtrace.

In some cases, the prepending and buffering options available in HP MPI 1.7's standard IO processing are useful in providing more readable output.

The default behavior is to print a stack trace.

Backtracing can be turned off entirely by setting the environment variable `MPI_NOBACKTRACE`. See"MPI_NOBACKTRACE" on page 45.

Backtracing is only supported on HP PA-RISC systems.

Troubleshooting HP MPI applications

This section describes limitations in HP MPI, some common difficulties you may face, and hints to help you overcome those difficulties and get the best performance from your HP MPI applications. Check this information first when you troubleshoot problems. The topics covered are organized by development task and also include answers to frequently asked questions:

- Building
- Starting
- Running
- Completing
- Frequently asked questions

To get information about the version of HP MPI installed on your system, use the what command. The following is an example of the command and its output:

```
% what /opt/mpi/bin/mpicc

/opt/mpi/bin/mpicc:

HP MPI 01.07.00.00 (dd/mm/yyyy) B6060BA - HP-UX 11.0
```

This command returns the HP MPI version number, the date this version was released, HP MPI product numbers, and the operating system version.

Building

You can solve most build-time problems by referring to the documentation for the compiler you are using.

If you use your own build script, specify all necessary input libraries. To determine what libraries are needed, check the contents of the compilation utilities stored in the HP MPI /opt/mpi/bin subdirectory.

HP MPI supports a 64-bit version of the MPI library on platforms running HP-UX 11.0. Both 32- and 64-bit versions of the library are shipped with HP-UX 11.0. For HP-UX 11.0, you cannot mix 32-bit and 64-bit executables in the same application.

HP MPI does not support Fortran applications that are compiled with the following options:

- `+autodblpad`— Fortran 77 programs
- `+autodbl`—Fortran 90 programs
- `+autodbl4`—Fortran 90 programs

Starting

CAUTION **Starting a MPI executable without the `mpirun` utility is no longer supported. For example, applications previously started by using `a.out -np # [args]` must now be started using `mpirun -np # a.out [args]`.**

When starting multihost applications, make sure that:

- All remote hosts are listed in your .rhosts file on each machine and you can `remsh` to the remote machines. The mpirun command has the `-ck` option you can use to determine whether the hosts and programs specified in your MPI application are available, and whether there are access or permission problems. Refer to “mpirun” on page 49.
- Application binaries are available on the necessary remote hosts and are executable on those machines
- The `-sp` option is passed to `mpirun` to set the target shell PATH environment variable. You can set this option in your appfile
- The .cshrc file does not contain `tty` commands such as `stty` if you are using a /bin/csh-based shell

Running

Run time problems originate from many sources and may include:

- Shared memory
- Message buffering
- Propagation of environment variables
- Interoperability
- Fortran 90 programming features
- UNIX open file descriptors
- External input and output

Shared memory

When an MPI application starts, each MPI process attempts to allocate a section of shared memory. This allocation can fail if the system-imposed limit on the maximum number of allowed shared-memory identifiers is exceeded or if the amount of available physical memory is not sufficient to fill the request.

After shared-memory allocation is done, every MPI process attempts to attach to the shared-memory region of every other process residing on the same host. This attachment can fail if the number of shared-memory segments attached to the calling process exceeds the system-imposed limit. In this case, use the `MPI_GLOBMEMSIZE` environment variable to reset your shared-memory allocation.

Furthermore, all processes must be able to attach to a shared-memory region at the same virtual address. For example, if the first process to attach to the segment attaches at address ADR, then the virtual-memory region starting at ADR must be available to all other processes. Placing `MPI_Init` to execute first can help avoid this problem. A process with a large stack size is also prone to this failure. Choose process stack size carefully.

Message buffering

According to the MPI standard, message buffering may or may not occur when processes communicate with each other using `MPI_Send`. `MPI_Send` buffering is at the discretion of the MPI implementation. Therefore, you should take care when coding communications that depend upon buffering to work correctly.

For example, when two processes use `MPI_Send` to simultaneously send a message to each other and use `MPI_Recv` to receive the messages, the results are unpredictable. If the messages are buffered, communication works correctly. If the messages are not buffered, however, each process hangs in `MPI_Send` waiting for `MPI_Recv` to take the message. For example, a sequence of operations (labeled "Deadlock") as illustrated in Table 9 would result in such a deadlock. Table 9 also illustrates the sequence of operations that would avoid code deadlock.

Table 9 Non-buffered messages and deadlock

Deadlock		**No Deadlock**	
Process 1	**Process 2**	**Process 1**	**Process 2**
MPI_Send(2,....)	MPI_Send(1,....)	MPI_Send(2,....)	MPI_Recv(1,....)
MPI_Recv(2,....)	MPI_Recv(1,....)	MPI_Recv(2,....)	MPI_Send(1,....)

Propagation of environment variables

When working with applications that run on multiple hosts, you must set values for environment variables on each host that participates in the job.

A recommended way to accomplish this is to set the `-e` option in the appfile:

`-h` *remote_host* `-e` *var=val* `[-np` *#*`]` *program* `[`*args*`]`

Refer to “Creating an appfile” on page 55 for details. Alternatively, you can set environment variables using the .cshrc file on each remote host if you are using a /bin/csh-based shell.

Interoperability

Depending upon what server resources are available, applications may run on heterogeneous systems.

For example, suppose you create an MPMD application that calculates the average acceleration of particles in a simulated cyclotron. The application consists of a four-process program called sum_accelerations and an eight-process program called calculate_average.

Because you have access to a K-Class server called K_server and an V-Class server called V_server, you create the following appfile:

```
-h K_server -np 4 sum_accelerations
-h V_server -np 8 calculate_average
```

Then, you invoke `mpirun` passing it the name of the appfile you created. Even though the two application programs run on different platforms, all processes can communicate with each other, resulting in twelve-way parallelism. The four processes belonging to the sum_accelerations application are ranked 0 through 3, and the eight processes belonging to the calculate_average application are ranked 4 through 11 because HP MPI assigns ranks in MPI_COMM_WORLD according to the order the programs appear in the appfile.

Fortran 90 programming features

The MPI 1.1 standard defines bindings for Fortran 77 but not Fortran 90.

Although most Fortran 90 MPI applications work using the Fortran 77 MPI bindings, some Fortran 90 features can cause unexpected behavior when used with HP MPI.

In Fortran 90, an array is not always stored in contiguous memory. When noncontiguous array data are passed to an HP MPI subroutine, Fortran 90 copies the data into temporary storage, passes it to the HP MPI subroutine, and copies it back when the subroutine returns. As a result, HP MPI is given the address of the copy but not of the original data.

In some cases, this copy-in and copy-out operation can cause a problem. For a nonblocking HP MPI call, the subroutine returns immediately and the temporary storage is deallocated. When HP MPI tries to access the already invalid memory, the behavior is unknown. Moreover, HP MPI operates close to the system level and needs to know the address of the original data. However, even if the address is known, HP MPI does not know if the data are contiguous or not.

UNIX open file descriptors

UNIX imposes a limit to the number of file descriptors that application processes can have open at one time. When running a multihost application, each local process opens a socket to each remote process. An HP MPI application with a large amount of off-host processes can quickly reach the file descriptor limit. Ask your system administrator to increase the limit if your applications frequently exceed the maximum.

External input and output

You can use stdin, stdout, and stderr in your applications to read and write data. All standard input is routed through the `mpirun` process. Standard input to `mpirun` is selectively ignored (default behavior), replicated to all of the MPI processes, or directed to a single process. Input intended for any of the processes in an MPI application should therefore be directed to the standard input of `mpirun`.

Since `mpirun` reads stdin on behalf of the processes, running an MPI application in the background will result in the application being suspended by most shells. For this reason, the default mode is to ignore stdin. If your application uses stdin, use the following options for making standard input available to processes.

Similarly, the stdout and stderr of MPI processes are combined to become the stdout and stderr of the `mpirun` process used to start the MPI application. How the streams are combined and displayed is determined by the MPI standard IO settings.

CAUTION **Applications that read from stdin must use -stdio=i or -stdio=i[n]**

HP MPI standard IO options can be set by using the following options to mpirun:

```
mpirun -stdio=[bline[#] | bnone[#] | b[#], [p], [r[#]],
[i[#]]
```

where

`i`	Broadcasts standard input to all MPI processes.
`i` *[#]*	Directs standard input to the process with global rank #.

The following modes are available for buffering:

`b` *[#>0]* Specifies that the output of a single MPI process is placed to the standard out of `mpirun` after # bytes of output have been accumulated.

`bnone` *[#>0]* The same as b[#] except that the buffer is flushed both when it is full and when it is found to contain any data. Essentially provides no buffering from the user's perspective.

`bline` *[#>0]* Displays the output of a process after a line feed is encountered, or the # byte buffer is full.

The default value of # in all cases is 10k bytes

The following option is available for prepending:

`p` Enables prepending. The global rank of the originating process is prepended to stdout and stderr output. Although this mode can be combined with any buffering mode, prepending makes the most sense with the modes `b` and `bline`.

The following option is available for combining repeated output:

`r` *[#>1]* Combines repeated identical output from the same process by prepending a multiplier to the beginning of the output. At most, # maximum repeated outputs are accumulated without display. This option is used only with `bline`. The default value of # is infinity.

Default: `-stdio=bline,i`

Completing

In HP MPI, `MPI_Finalize` is a barrier-like collective routine that waits until all application processes have called it before returning. If your application exits without calling `MPI_Finalize`, pending requests may not complete.

When running an application, `mpirun` waits until all processes have exited. If an application detects an MPI error that leads to program termination, it calls `MPI_Abort` instead.

You may want to code your error conditions using `MPI_Abort`, which cleans up the application.

Each HP MPI application is identified by a job ID, unique on the server where `mpirun` is invoked. If you use the `-j` option, `mpirun` prints the job ID of the application that it runs. Then, you can invoke `mpijob` with the job ID to display the status of your application.

If your application hangs or terminates abnormally, you can use `mpiclean` to kill any lingering processes and shared-memory segments. `mpiclean` uses the job ID from `mpirun -j` to specify the application to terminate.

Frequently asked questions

This section describes frequently asked HP MPI questions. These questions address the following issues:

- Time in MPI_Finalize
- MPI clean up
- Application hangs in MPI_Send

Time in MPI_Finalize

QUESTION: When I build with HP MPI and then turn tracing on, the application takes a long time inside `MPI_Finalize`. What is causing this?

ANSWER: When you turn tracing on `MPI_Finalize` spends time consolidating the raw trace generated by each process into a single output file (with a .tr extension).

MPI clean up

QUESTION: How does HP MPI clean up when something goes wrong?

ANSWER: HP MPI uses several mechanisms to clean up job files. Note that all processes in your application must call `MPI_Finalize`.

- When a correct HP MPI program (that is, one that calls `MPI_Finalize`) exits successfully, the root host deletes the job file.
- If you use `mpirun`, it deletes the job file when the application terminates, whether successfully or not.
- When an application calls `MPI_Abort`, `MPI_Abort` deletes the job file.
- If you use `mpijob -j` to get more information on a job, and the processes of that job have all exited, `mpijob` issues a warning that the job has completed, and deletes the job file.

Application hangs in MPI_Send

QUESTION: My MPI application hangs at MPI_Send. Why?

ANSWER: Deadlock situations can occur when your code uses standard send operations and assumes buffering behavior for standard communication mode. You should not assume message buffering between processes because the MPI standard does not mandate a buffering strategy. HP MPI does sometimes use buffering for MPI_Send and MPI_Rsend, but it is dependent on message size and at the discretion of the implementation.

QUESTION: How can I tell if the deadlock is because my code depends on buffering?

ANSWER: To quickly determine whether the problem is due to your code being dependent on buffering, set the z option for MPI_FLAGS. MPI_FLAGS modifies the general behavior of HP MPI, and in this case converts MPI_Send and MPI_Rsend calls in your code to MPI_Ssend, without you having to rewrite your code. MPI_Ssend guarantees synchronous send semantics, that is, a send can be started whether or not a matching receive is posted. However, the send completes successfully only if a matching receive is posted and the receive operation has started to receive the message sent by the synchronous send.

If your application still hangs after you convert MPI_Send and MPI_Rsend calls to MPI_Ssend, you know that your code is written to depend on buffering. You should rewrite it so that MPI_Send and MPI_Rsend do not depend on buffering.

Alternatively, use nonblocking communication calls to initiate send operations. A nonblocking send-start call returns before the message is copied out of the send buffer, but a separate send-complete call is needed to complete the operation. Refer also to “Sending and receiving messages” on page 6 for information about blocking and nonblocking communication. Refer to “MPI_FLAGS” on page 37 for information about MPI_FLAGS options.

A Example applications

This appendix provides example applications that supplement the conceptual information throughout the rest of this book about MPI in general and HP MPI in particular. Table 10 summarizes the examples in this appendix. The example codes are also included in the /opt/mpi/help subdirectory in your HP MPI product.

Table 10 Example applications shipped with HP MPI

Name	Language	Description	-np argument
send_receive.f	Fortran 77	Illustrates a simple send and receive operation.	-np >= 2
ping_pong.c	C	Measures the time it takes to send and receive data between two processes.	-np = 2
compute_pi.f	Fortran 77	Computes pi by integrating $f(x)=4/(1+x^2)$.	-np >= 1
master_worker.f90	Fortran 90	Distributes sections of an array and does computation on all sections in parallel.	-np >= 2
cart.C	C++	Generates a virtual topology.	-np = 4
communicator.c	C	Copies the default communicator MPI_COMM_WORLD.	-np = 2
multi_par.f	Fortran 77	Uses the alternating direction iterative (ADI) method on a 2-dimensional compute region.	-np >= 1
io.c	C	Writes data for each process to a separate file called iodata*x*, where *x* represents each process rank in turn. Then, the data in iodata*x* is read back.	-np >= 1
thread_safe.c	C	Tracks the number of client requests handled and prints a log of the requests to stdout.	-np >= 2

These examples and the Makefile are located in the /opt/mpi/help subdirectory. The examples are presented for illustration purposes only. They may not necessarily represent the most efficient way to solve a given problem.

To build and run the examples follow the following procedure:

Step 1. Change to a writable directory.

Step 2. Copy all files from the help directory to the current writable directory:

```
% cp /opt/mpi/help/* .
```

Step 3. Compile all the examples or a single example.

To compile and run all the examples in the /help directory, at your UNIX prompt enter:

```
% make
```

To compile and run the thread_safe.c program only, at your UNIX prompt enter:

```
% make thread_safe
```

send_receive.f

In this Fortran 77 example, process 0 sends an array to other processes in the default communicator MPI_COMM_WORLD.

```
program main

include 'mpif.h'

integer rank, size, to, from, tag, count, i, ierr
integer src, dest
integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE)
double precision data(100)

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

if (size .eq. 1) then
        print *, 'must have at least 2 processes'
        call MPI_Finalize(ierr)
        stop
endif

print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0

if (rank .eq. src) then
        to = dest
        count = 10
        tag = 2001

        do i=1, 10
                 data(i) = 1
        enddo

        call MPI_Send(data, count, MPI_DOUBLE_PRECISION,
+                       to, tag, MPI_COMM_WORLD, ierr)
endif

if (rank .eq. dest) then
         tag = MPI_ANY_TAG
         count = 10
         from = MPI_ANY_SOURCE
         call MPI_Recv(data, count, MPI_DOUBLE_PRECISION,
+                       from, tag, MPI_COMM_WORLD, status, ierr)
```

```
call MPI_Get_Count(status, MPI_DOUBLE_PRECISION,
+                          st_count, ierr)
st_source = status(MPI_SOURCE)
st_tag = status(MPI_TAG)

print *, 'Status info: source = ', st_source,
+                ' tag = ', st_tag, ' count = ', st_count
print *, rank, ' received', (data(i),i=1,10)

endif

call MPI_Finalize(ierr)
stop
end
```

send_receive output

The output from running the send_receive executable is shown below. The application was run with `-np` = 10.

```
Process  0 of  10 is alive
Process  1 of  10 is alive
Process  3 of  10 is alive
Process  5 of  10 is alive
Process  9 of  10 is alive
Process  2 of  10 is alive
Status info: source =  0 tag =  2001 count =  10
9 received 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
Process  4 of  10 is alive
Process  7 of  10 is alive
Process  8 of  10 is alive
Process  6 of  10 is alive
```

ping_pong.c

This C example is used as a performance benchmark to measure the amount of time it takes to send and receive data between two processes. The buffers are aligned and offset from each other to avoid cache conflicts caused by direct process-to-process byte-copy operations

To run this example:

- Define the CHECK macro to check data integrity.
- Increase the number of bytes to at least twice the cache size to obtain representative bandwidth measurements.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define NLOOPS          1000
#define ALIGN           4096

main(argc, argv)

int                     argc;
char                    *argv[];

{
        int             i, j;
        double          start, stop;
        int             nbytes = 0;
        int             rank, size;
        MPI_Status      status;
        char            *buf;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (size != 2) {

                    if ( ! rank) printf("ping_pong: must have two processes\n");
                    MPI_Finalize();
                    exit(0);
        }

        nbytes = (argc > 1) ? atoi(argv[1]) : 0;
        if (nbytes < 0) nbytes = 0;
```

ping_pong.c

```
/*
* Page-align buffers and displace them in the cache to avoid collisions.
*/

        buf = (char *) malloc(nbytes + 524288 + (ALIGN - 1));
        if (buf == 0) {
                    MPI_Abort(MPI_COMM_WORLD, MPI_ERR_BUFFER);
                    exit(1);
        }

        buf = (char *) ((((unsigned long) buf) + (ALIGN - 1)) & ~(ALIGN - 1));
        if (rank == 1) buf += 524288;
        memset(buf, 0, nbytes);

/*
* Ping-pong.
*/

        if (rank == 0) {
                    printf("ping-pong %d bytes ...\n", nbytes);
/*
* warm-up loop
*/

                    for (i = 0; i < 5; i++) {
                         MPI_Send(buf, nbytes, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
                         MPI_Recv(buf, nbytes, MPI_CHAR,1, 1,
                                                  MPI_COMM_WORLD, &status);
                    }
/*
* timing loop
*/

        start = MPI_Wtime();
        for (i = 0; i < NLOOPS; i++) {
#ifdef CHECK
                    for (j = 0; j < nbytes; j++) buf[j] = (char) (j + i);
#endif
                    MPI_Send(buf, nbytes, MPI_CHAR,1, 1000 + i, MPI_COMM_WORLD);
#ifdef CHECK
                    memset(buf, 0, nbytes);
#endif
                    MPI_Recv(buf, nbytes, MPI_CHAR,1, 2000 + i,
                                               MPI_COMM_WORLD,&status);
```

```
#ifdef CHECK

                  for (j = 0; j < nbytes; j++) {
                       if (buf[j] != (char) (j + i)) {
                              printf("error: buf[%d] = %d, not %d\n",j,
                                    buf[j], j + i);
                              break;
                       }
                   }
#endif
         }
          stop = MPI_Wtime();

           printf("%d bytes: %.2f usec/msg\n",
                      nbytes, (stop - start) / NLOOPS / 2 * 1000000);

            if (nbytes > 0) {
                    printf("%d bytes: %.2f MB/sec\n", nbytes,nbytes / 1000000./
                                        ((stop - start) / NLOOPS / 2));
             }
          }
          else {
/*
* warm-up loop
*/
          for (i = 0; i < 5; i++) {
                    MPI_Recv(buf, nbytes, MPI_CHAR,0, 1, MPI_COMM_WORLD, &status);
                    MPI_Send(buf, nbytes, MPI_CHAR, 0, 1, MPI_COMM_WORLD);
           }

           for (i = 0; i < NLOOPS; i++) {
                    MPI_Recv(buf, nbytes, MPI_CHAR,0, 1000 + i,
                                                MPI_COMM_WORLD,&status);
                    MPI_Send(buf, nbytes, MPI_CHAR,0, 2000 + i, MPI_COMM_WORLD);
            }
        }
        MPI_Finalize();
        exit(0);
}
```

ping_pong output

The output from running the ping_pong executable is shown below. The application was run with `-np = 2`.

```
ping-pong 0 bytes ...
0 bytes: 2.98 3.99 34.99 usec/msg
```

compute_pi.f

This Fortran 77 example computes pi by integrating $f(x) = 4/(1 + x^2)$. Each process:

- Receives the number of intervals used in the approximation
- Calculates the areas of its rectangles
- Synchronizes for a global summation

Process 0 prints the result of the calculation.

```
program main

      include 'mpif.h'

      double precision PI25DT
      parameter(PI25DT = 3.14159265358979323846264338327950288419716939937510d0)

      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr
C
C Function to integrate
C
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
      print *, "Process ", myid, " of ", numprocs, " is alive"

      sizetype = 1
      sumtype = 2

      if (myid .eq. 0) then
              n = 100
      endif

      call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
C
C Calculate the interval size.
C
      h = 1.0d0 / n
      sum  = 0.0d0

      do 20 i = myid + 1, n, numprocs
              x = h * (dble(i) - 0.5d0)
              sum = sum + f(x)
 20   continue
```

```
      mypi = h * sum
C
C Collect all the partial sums.
C
      call MPI_REDUCE(mypi, pi, 1, MPI_DOUBLE_PRECISION,
     +                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
C
C Process 0 prints the result.
C
      if (myid .eq. 0) then
              write(6, 97) pi, abs(pi - PI25DT)
 97           format('  pi is approximately: ', F18.16,
     +               '  Error is: ', F18.16)
      endif

      call MPI_FINALIZE(ierr)

      stop
      end
```

compute_pi output

The output from running the compute_pi executable is shown below. The application was run with `-np` = 10.

```
Process  0 of  10 is alive
Process  1 of  10 is alive
Process  3 of  10 is alive
Process  9 of  10 is alive
Process  7 of  10 is alive
Process  5 of  10 is alive
Process  6 of  10 is alive
Process  2 of  10 is alive
Process  4 of  10 is alive
Process  8 of  10 is alive
pi is approximately: 3.1416009869231250
Error is:  .0000083333333318
```

master_worker.f90

In this Fortran 90 example, a master task initiates (numtasks - 1) number of worker tasks. The master distributes an equal portion of an array to each worker task. Each worker task receives its portion of the array and sets the value of each element to (the element's index + 1). Each worker task then sends its portion of the modified array back to the master.

```
program array_manipulation
  include 'mpif.h'

  integer (kind=4) :: status(MPI_STATUS_SIZE)
  integer (kind=4), parameter :: ARRAYSIZE = 10000, MASTER = 0
  integer (kind=4) :: numtasks, numworkers, taskid, dest, index, i
  integer (kind=4) :: arraymsg, indexmsg, source, chunksize, int4, real4
  real (kind=4) :: data(ARRAYSIZE), result(ARRAYSIZE)
  integer (kind=4) :: numfail, ierr

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD, taskid, ierr)
  call MPI_Comm_size(MPI_COMM_WORLD, numtasks, ierr)
  numworkers = numtasks - 1
  chunksize = (ARRAYSIZE / numworkers)
  arraymsg = 1
  indexmsg = 2
  int4 = 4
  real4 = 4
  numfail = 0

! ****************************** Master task ******************************
  if (taskid .eq. MASTER) then
     data = 0.0
     index = 1
     do dest = 1, numworkers
        call MPI_Send(index, 1, MPI_INTEGER, dest, 0, MPI_COMM_WORLD, ierr)
        call MPI_Send(data(index), chunksize, MPI_REAL, dest, 0, &
             MPI_COMM_WORLD, ierr)
        index = index + chunksize
     end do

     do i = 1, numworkers
        source = i
        call MPI_Recv(index, 1, MPI_INTEGER, source, 1, MPI_COMM_WORLD, &
             status, ierr)
        call MPI_Recv(result(index), chunksize, MPI_REAL, source, 1, &
             MPI_COMM_WORLD, status, ierr)
     end do
```

```
      do i = 1, numworkers*chunksize
         if (result(i) .ne. (i+1)) then
            print *, 'element ', i, ' expecting ', (i+1), ' actual is ', result(i)
            numfail = numfail + 1
         endif
      enddo

      if (numfail .ne. 0) then
         print *, 'out of ', ARRAYSIZE, ' elements, ', numfail, ' wrong answers'
      else
         print *, 'correct results!'
      endif
   end if

! **************************** Worker task ******************************
   if (taskid .gt. MASTER) then
      call MPI_Recv(index, 1, MPI_INTEGER, MASTER, 0, MPI_COMM_WORLD, &
           status, ierr)
      call MPI_Recv(result(index), chunksize, MPI_REAL, MASTER, 0, &
           MPI_COMM_WORLD, status, ierr)

      do i = index, index + chunksize - 1
         result(i) = i + 1
      end do

      call MPI_Send(index, 1, MPI_INTEGER, MASTER, 1, MPI_COMM_WORLD, ierr)
      call MPI_Send(result(index), chunksize, MPI_REAL, MASTER, 1, &
           MPI_COMM_WORLD, ierr)
   end if

   call MPI_Finalize(ierr)

end program array_manipulation
```

master_worker output

The output from running the master_worker executable is shown below. The application was run with `-np = 2`.

```
correct results!
```

cart.C

This C++ program generates a virtual topology. The class Node represents a node in a 2-D torus. Each process is assigned a node or nothing. Each node holds integer data, and the shift operation exchanges the data with its neighbors. Thus, north-east-south-west shifting returns the initial data.

```
#include <stdio.h>
#include <mpi.h>

#define NDIMS   2

typedef enum { NORTH, SOUTH, EAST, WEST } Direction;

// A node in 2-D torus
class Node {
private:
  MPI_Comm      comm;
  int           dims[NDIMS], coords[NDIMS];
  int           grank, lrank;
  int           data;
public:
  Node(void);
  ~Node(void);
  void profile(void);
  void print(void);
  void shift(Direction);
};

// A constructor
Node::Node(void)
{
  int i, nnodes, periods[NDIMS];

  // Create a balanced distribution
  MPI_Comm_size(MPI_COMM_WORLD, &nnodes);
  for (i = 0; i < NDIMS; i++) { dims[i] = 0; }
  MPI_Dims_create(nnodes, NDIMS, dims);

  // Establish a cartesian topology communicator
  for (i = 0; i < NDIMS; i++) { periods[i] = 1; }
  MPI_Cart_create(MPI_COMM_WORLD, NDIMS, dims, periods, 1, &comm);

 // Initialize the data
  MPI_Comm_rank(MPI_COMM_WORLD, &grank);
  if (comm == MPI_COMM_NULL) {
    lrank = MPI_PROC_NULL;
    data = -1;
```

```
  } else {
    MPI_Comm_rank(comm, &lrank);
    data = lrank;
    MPI_Cart_coords(comm, lrank, NDIMS, coords);
  }
}

// A destructor
Node::~Node(void)
{
  if (comm != MPI_COMM_NULL) {
    MPI_Comm_free(&comm);
  }
}

// Shift function
void Node::shift(Direction dir)
{
  if (comm == MPI_COMM_NULL) { return; }

  int direction, disp, src, dest;

  if (dir == NORTH) {
    direction = 0; disp = -1;
  } else if (dir == SOUTH) {
    direction = 0; disp = 1;
  } else if (dir == EAST) {
    direction = 1; disp = 1;
  } else {
    direction = 1; disp = -1;
  }
  MPI_Cart_shift(comm, direction, disp, &src, &dest);
  MPI_Status stat;
  MPI_Sendrecv_replace(&data, 1, MPI_INT, dest, 0, src, 0, comm, &stat);
}
// Synchronize and print the data being held

void Node::print(void)
{
  if (comm != MPI_COMM_NULL) {
    MPI_Barrier(comm);
    if (lrank == 0) { puts(""); } // line feed
    MPI_Barrier(comm);
    printf("(%d, %d) holds %d\n", coords[0], coords[1], data);
  }
}

// Print object's profile
void Node::profile(void)
{
  // Non-member does nothing
  if (comm == MPI_COMM_NULL) { return; }
```

cart.C

```
  // Print "Dimensions" at first
  if (lrank == 0) {
    printf("Dimensions: (%d, %d)\n", dims[0], dims[1]);
  }
  MPI_Barrier(comm);

  // Each process prints its profile
  printf("global rank %d: cartesian rank %d, coordinate (%d, %d)\n",
         grank, lrank, coords[0], coords[1]);
}

// Program body
//
// Define a torus topology and demonstrate shift operations.
//

void body(void)
{
  Node node;

  node.profile();

  node.print();

  node.shift(NORTH);
  node.print();
  node.shift(EAST);
  node.print();
  node.shift(SOUTH);
  node.print();
  node.shift(WEST);
  node.print();
}
//
// Main program---it is probably a good programming practice to call
//   MPI_Init() and MPI_Finalize() here.
//
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  body();
  MPI_Finalize();
}
```

cart output

The output from running the cart executable is shown below. The application was run with -np = 4.

```
Dimensions: (2, 2)
global rank 0: cartesian rank 0, coordinate (0, 0)
global rank 2: cartesian rank 2, coordinate (1, 0)
global rank 3: cartesian rank 3, coordinate (1, 1)
global rank 1: cartesian rank 1, coordinate (0, 1)

(0, 0) holds 0
(0, 1) holds 1
(1, 0) holds 2
(1, 1) holds 3

(0, 0) holds 2
(0, 1) holds 3
(1, 0) holds 0
(1, 1) holds 1

(0, 0) holds 3
(0, 1) holds 2
(1, 0) holds 1
(1, 1) holds 0

(0, 0) holds 1
(0, 1) holds 0
(1, 0) holds 3
(1, 1) holds 2

(0, 0) holds 0
(1, 1) holds 3
(1, 0) holds 2
(0, 1) holds 1
```

communicator.c

This C example shows how to make a copy of the default communicator MPI_COMM_WORLD using MPI_Comm_dup.

```
#include <stdio.h>
#include <mpi.h>

main(argc, argv)

int                     argc;
char                    *argv[];

{
        int             rank, size, data;
        MPI_Status      status;
        MPI_Comm        libcomm;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if (size != 2) {
                if ( ! rank) printf("communicator: must have two processes\n");
                MPI_Finalize();
                exit(0);
        }

        MPI_Comm_dup(MPI_COMM_WORLD, &libcomm);

        if (rank == 0) {
                data = 12345;
                MPI_Send(&data, 1, MPI_INT, 1, 5, MPI_COMM_WORLD);
                data = 6789;
                MPI_Send(&data, 1, MPI_INT, 1, 5, libcomm);
        } else {
                MPI_Recv(&data, 1, MPI_INT, 0, 5, libcomm, &status);
                printf("received libcomm data = %d\n", data);
                MPI_Recv(&data, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &status);
                printf("received data = %d\n", data);
        }

        MPI_Comm_free(&libcomm);
        MPI_Finalize();
        exit(0);
}
```

communicator output

The output from running the communicator executable is shown below. The application was run with `-np` = 2.

```
received libcomm data = 6789
received data = 12345
```

multi_par.f

The Alternating Direction Iterative (ADI) method is often used to solve differential equations. In this example, multi_par.f, a compiler that supports OPENMP directives is required in order to achieve multi-level parallelism.

multi_par.f implements the following logic for a 2-dimensional compute region:

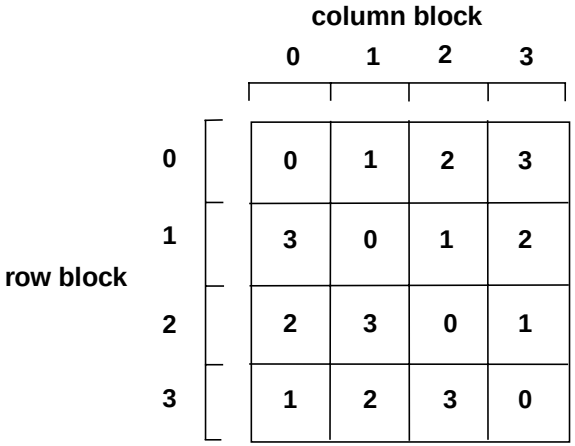
```
DO J=1,JMAX
  DO I=2,IMAX
    A(I,J)=A(I,J)+A(I-1,J)
  ENDDO
ENDDO

DO J=2,JMAX
  DO I=1,IMAX
    A(I,J)=A(I,J)+A(I,J-1)
  ENDDO
ENDDO
```

There are loop-carried dependencies in the first inner DO loop (the array's rows) and the second outer DO loop (the array's columns).

Partitioning the array into column sections supports parallelization of the first outer loop. Partitioning the array into row sections supports parallelization of the second outer loop. However, this approach requires a massive data exchange among processes because of run-time partition changes.

In this case, twisted-data layout partitioning is a better approach because the partitioning used for the parallelization of the first outer loop can accommodate the partitioning of the second outer loop. The partitioning of the array is shown in Figure 20.

Figure 20 Array partitioning



In this sample program, the rank *n* process is assigned to the partition *n* at distribution initialization. Because these partitions are not contiguous-memory regions, MPI's derived datatype is used to define the partition layout to the MPI system.

Each process starts with computing summations in row-wise fashion. For example, the rank 2 process starts with the block that is on the 0th-row block and 2nd-column block (denoted as [0,2]).

The block computed in the second step is [1,3]. Computing the first row elements in this block requires the last row elements in the [0,3] block (computed in the first step in the rank 3 process). Thus, the rank 2 process receives the data from the rank 3 process at the beginning of the second step. Note that the rank 2 process also sends the last row elements of the [0,2] block to the rank 1 process that computes [1,2] in the second step. By repeating these steps, all processes finish summations in row-wise fashion (the first outer-loop in the illustrated program).

The second outer-loop (the summations in column-wise fashion) is done in the same manner. For example, at the beginning of the second step for the column-wise summations, the rank 2 process receives data from the rank 1 process that computed the [3,0] block. The rank 2 process also sends the last column of the [2,0] block to the rank 3 process. Note that each process keeps the same blocks for both of the outer-loop computations.

This approach is good for distributed memory architectures on which repartitioning requires massive data communications that are expensive. However, on shared memory architectures, the partitioning of the compute region does not imply data distribution. The row- and column-block partitioning method requires just one synchronization at the end of each outer loop.

For distributed shared-memory architectures, the mix of the two methods can be effective. The sample program implements the twisted-data layout method with MPI and the row- and column-block partitioning method with OPENMP thread directives. In the first case, the data dependency is easily satisfied as each thread computes down a different set of columns. In the second case we still want to compute down the columns for cache reasons, but to satisfy the data dependency, each thread computes a different portion of the same column and the threads work left to right across the rows together.

```
 implicit none

 include 'mpif.h'

 integer nrow                      ! # of rows

 integer ncol                      ! # of columns

 parameter(nrow=1000,ncol=1000)

 double precision array(nrow,ncol) ! compute region

 integer blk                       ! block iteration counter
 integer rb                        ! row block number
 integer cb                        ! column block number
 integer nrb                       ! next row block number
 integer ncb                       ! next column block number
 integer rbs(:)                    ! row block start subscripts
 integer rbe(:)                    ! row block end subscripts
 integer cbs(:)                    ! column block start subscripts
 integer cbe(:)                    ! column block end subscripts
 integer rdtype(:)                 ! row block communication datatypes
 integer cdtype(:)                 ! column block communication datatypes
 integer twdtype(:)                ! twisted distribution datatypes
 integer ablen(:)                  ! array of block lengths
 integer adisp(:)                  ! array of displacements
 integer adtype(:)                 ! array of datatypes
 allocatable rbs,rbe,cbs,cbe,rdtype,cdtype,twdtype,ablen,adisp,
*     adtype
 integer rank                      ! rank iteration counter
 integer comm_size                 ! number of MPI processes
 integer comm_rank                 ! sequential ID of MPI process
 integer ierr                      ! MPI error code
 integer mstat(mpi_status_size)    ! MPI function status
 integer src                       ! source rank
 integer dest                      ! destination rank
 integer dsize                     ! size of double precision in bytes
```

```
double precision startt,endt,elapsed ! time keepers
external compcolumn,comprow       ! subroutines execute in threads

c
c     MPI initialization
c
      call mpi_init(ierr)
      call mpi_comm_size(mpi_comm_world,comm_size,ierr)
      call mpi_comm_rank(mpi_comm_world,comm_rank,ierr)

c
c     Data initialization and start up
c
      if (comm_rank.eq.0) then
         write(6,*) 'Initializing',nrow,' x',ncol,' array...'
         call getdata(nrow,ncol,array)
         write(6,*) 'Start computation'
      endif
      call mpi_barrier(MPI_COMM_WORLD,ierr)
      startt=mpi_wtime()
c
c     Compose MPI datatypes for row/column send-receive
c
c     Note that the numbers from rbs(i) to rbe(i) are the indices
c     of the rows belonging to the i'th block of rows.  These indices
c     specify a portion (the i'th portion) of a column and the
c     datatype rdtype(i) is created as an MPI contiguous datatype
c     to refer to the i'th portion of a column.  Note this is a
c     contiguous datatype because fortran arrays are stored
c     column-wise.
c
c     For a range of columns to specify portions of rows, the situation
c     is similar:  the numbers from cbs(j) to cbe(j) are the indices
c     of the columns belonging to the j'th block of columns.  These
c     indices specify a portion (the j'th portion) of a row, and the
c     datatype cdtype(j) is created as an MPI vector datatype to refer
c     to the j'th portion of a row.  Note this a vector datatype
c     because adjacent elements in a row are actually spaced nrow
c     elements apart in memory.
c
      allocate(rbs(0:comm_size-1),rbe(0:comm_size-1),cbs(0:comm_size-1),
     *     cbe(0:comm_size-1),rdtype(0:comm_size-1),
     *     cdtype(0:comm_size-1),twdtype(0:comm_size-1))
      do blk=0,comm_size-1
         call blockasgn(1,nrow,comm_size,blk,rbs(blk),rbe(blk))
         call mpi_type_contiguous(rbe(blk)-rbs(blk)+1,
     *        mpi_double_precision,rdtype(blk),ierr)
         call mpi_type_commit(rdtype(blk),ierr)
         call blockasgn(1,ncol,comm_size,blk,cbs(blk),cbe(blk))
         call mpi_type_vector(cbe(blk)-cbs(blk)+1,1,nrow,
     *        mpi_double_precision,cdtype(blk),ierr)
         call mpi_type_commit(cdtype(blk),ierr)
      enddo
```

```
c     Compose MPI datatypes for gather/scatter
c
c     Each block of the partitioning is defined as a set of fixed length
c     vectors.  Each process'es partition is defined as a struct of such
c     blocks.
c
      allocate(adtype(0:comm_size-1),adisp(0:comm_size-1),
     *     ablen(0:comm_size-1))
      call mpi_type_extent(mpi_double_precision,dsize,ierr)
      do rank=0,comm_size-1
         do rb=0,comm_size-1
            cb=mod(rb+rank,comm_size)
            call mpi_type_vector(cbe(cb)-cbs(cb)+1,rbe(rb)-rbs(rb)+1,
     *           nrow,mpi_double_precision,adtype(rb),ierr)
            call mpi_type_commit(adtype(rb),ierr)
            adisp(rb)=((rbs(rb)-1)+(cbs(cb)-1)*nrow)*dsize
            ablen(rb)=1
         enddo
         call mpi_type_struct(comm_size,ablen,adisp,adtype,
     *        twdtype(rank),ierr)
         call mpi_type_commit(twdtype(rank),ierr)
         do rb=0,comm_size-1
            call mpi_type_free(adtype(rb),ierr)
         enddo
      enddo
      deallocate(adtype,adisp,ablen)

c     Scatter initial data with using derived datatypes defined above
c     for the partitioning.  MPI_send() and MPI_recv() will find out the
c     layout of the data from those datatypes.  This saves application
c     programs to manually pack/unpack the data, and more importantly,
c     gives opportunities to the MPI system for optimal communication
c     strategies.
c
      if (comm_rank.eq.0) then
         do dest=1,comm_size-1
            call mpi_send(array,1,twdtype(dest),dest,0,mpi_comm_world,
     *           ierr)
         enddo
      else
         call mpi_recv(array,1,twdtype(comm_rank),0,0,mpi_comm_world,
     *        mstat,ierr)
      endif

c
c     Computation
c
c     Sum up in each column.
c     Each MPI process, or a rank, computes blocks that it is assigned.
c     The column block number is assigned in the variable 'cb'.  The
c     starting and ending subscripts of the column block 'cb' are
c     stored in 'cbs(cb)' and 'cbe(cb)', respectively.  The row block
c     number is assigned in the variable 'rb'.  The starting and ending
c     subscripts of the row block 'rb' are stored in 'rbs(rb)' and
c     'rbe(rb)', respectively, as well.
      src=mod(comm_rank+1,comm_size)
```

```
      dest=mod(comm_rank-1+comm_size,comm_size)
      ncb=comm_rank
      do rb=0,comm_size-1
         cb=ncb
c
c     Compute a block.  The function will go thread-parallel if the
c     compiler supports OPENMP directives.
c
         call compcolumn(nrow,ncol,array,
     *                   rbs(rb),rbe(rb),cbs(cb),cbe(cb))
         if (rb.lt.comm_size-1) then
c
c     Send the last row of the block to the rank that is to compute the
c     block next to the computed block.  Receive the last row of the
c     block that the next block being computed depends on.
c
            nrb=rb+1
            ncb=mod(nrb+comm_rank,comm_size)
            call mpi_sendrecv(array(rbe(rb),cbs(cb)),1,cdtype(cb),dest,
     *           0,array(rbs(nrb)-1,cbs(ncb)),1,cdtype(ncb),src,0,
     *           mpi_comm_world,mstat,ierr)
         endif
      enddo
c
c     Sum up in each row.
c     The same logic as the loop above except rows and columns are
c     switched.
c
      src=mod(comm_rank-1+comm_size,comm_size)
      dest=mod(comm_rank+1,comm_size)
      do cb=0,comm_size-1
         rb=mod(cb-comm_rank+comm_size,comm_size)
         call comprow(nrow,ncol,array,
     *                rbs(rb),rbe(rb),cbs(cb),cbe(cb))
         if (cb.lt.comm_size-1) then
            ncb=cb+1
            nrb=mod(ncb-comm_rank+comm_size,comm_size)
            call mpi_sendrecv(array(rbs(rb),cbe(cb)),1,rdtype(rb),dest,
     *           0,array(rbs(nrb),cbs(ncb)-1),1,rdtype(nrb),src,0,
     *           mpi_comm_world,mstat,ierr)
         endif
      enddo
c
c     Gather computation results
c
      call mpi_barrier(MPI_COMM_WORLD,ierr)
      endt=mpi_wtime()

      if (comm_rank.eq.0) then
         do src=1,comm_size-1
            call mpi_recv(array,1,twdtype(src),src,0,mpi_comm_world,
     *           mstat,ierr)
         enddo
```

```
         elapsed=endt-startt
         write(6,*) 'Computation took',elapsed,' seconds'
      else
         call mpi_send(array,1,twdtype(comm_rank),0,0,mpi_comm_world,
     *        ierr)
      endif
c
c     Dump to a file
c
c      if (comm_rank.eq.0) then
c         print*,'Dumping to adi.out...'
c         open(8,file='adi.out')
c         write(8,*) array
c         close(8,status='keep')
c      endif
c
c     Free the resources
c
      do rank=0,comm_size-1
         call mpi_type_free(twdtype(rank),ierr)
      enddo
      do blk=0,comm_size-1
         call mpi_type_free(rdtype(blk),ierr)
         call mpi_type_free(cdtype(blk),ierr)
      enddo
      deallocate(rbs,rbe,cbs,cbe,rdtype,cdtype,twdtype)
c
c     Finalize the MPI system
c
      call mpi_finalize(ierr)
      end
c**********************************************************************
      subroutine blockasgn(subs,sube,blockcnt,nth,blocks,blocke)
c
c     This subroutine:
c     is given a range of subscript and the total number of blocks in
c     which the range is to be divided, assigns a subrange to the caller
c     that is n-th member of the blocks.
c
      implicit none
      integer subs           ! (in)      subscript start
      integer sube           ! (in)      subscript end
      integer blockcnt       ! (in)      block count
      integer nth            ! (in)      my block (begin from 0)
      integer blocks         ! (out)     assigned block start subscript
      integer blocke         ! (out)     assigned block end subscript
c
      integer d1,m1
c
      d1=(sube-subs+1)/blockcnt
      m1=mod(sube-subs+1,blockcnt)
      blocks=nth*d1+subs+min(nth,m1)
      blocke=blocks+d1-1
      if(m1.gt.nth)blocke=blocke+1
```

```
      end
c
c*********************************************************************
      subroutine compcolumn(nrow,ncol,array,rbs,rbe,cbs,cbe)
c
c     This subroutine:
c     does summations of columns in a thread.
c
      implicit none

      integer nrow                       ! # of rows
      integer ncol                       ! # of columns
      double precision array(nrow,ncol)  ! compute region
      integer rbs                        ! row block start subscript
      integer rbe                        ! row block end subscript
      integer cbs                        ! column block start subscript
      integer cbe                        ! column block end subscript

c
c     Local variables
c
      integer i,j

c
c     The OPENMP directive below allows the compiler to split the
c     values for "j" between a number of threads.  By making i and j
c     private, each thread works on its own range of columns "j",
c     and works down each column at its own pace "i".
c
c     Note no data dependency problems arise by having the threads all
c     working on different columns simultaneously.
c

C$OMP PARALLEL DO PRIVATE(i,j)
      do j=cbs,cbe
         do i=max(2,rbs),rbe
            array(i,j)=array(i-1,j)+array(i,j)
         enddo
      enddo
C$OMP END PARALLEL DO
      end

c*********************************************************************
      subroutine comprow(nrow,ncol,array,rbs,rbe,cbs,cbe)
c
c     This subroutine:
c     does summations of rows in a thread.
c
      implicit none

      integer nrow                       ! # of rows
      integer ncol                       ! # of columns
      double precision array(nrow,ncol)  ! compute region
      integer rbs                        ! row block start subscript
      integer rbe                        ! row block end subscript
      integer cbs                        ! column block start subscript
```

```
      integer cbe                       ! column block end subscript

c
c     Local variables
c
      integer i,j

c
c     The OPENMP directives below allow the compiler to split the
c     values for "i" between a number of threads, while "j" moves
c     forward lock-step between the threads.  By making j shared
c     and i private, all the threads work on the same column "j" at
c     any given time, but they each work on a different portion "i"
c     of that column.
c
c     This is not as efficient as found in the compcolumn subroutine,
c     but is necessary due to data dependencies.
c


C$OMP PARALLEL PRIVATE(i)
      do j=max(2,cbs),cbe
C$OMP DO
         do i=rbs,rbe
            array(i,j)=array(i,j-1)+array(i,j)
         enddo
C$OMP END DO
      enddo
C$OMP END PARALLEL

      end
c
c**********************************************************************

      subroutine getdata(nrow,ncol,array)
c
c     Enter dummy data
c
      integer nrow,ncol
      double precision array(nrow,ncol)
c
      do j=1,ncol
         do i=1,nrow
            array(i,j)=(j-1.0)*ncol+i
         enddo
      enddo
      end
```

io.c

In this C example, each process writes to a separate file called iodata*x*, where *x* represents each process rank in turn. Then, the data in iodata*x* is read back.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <mpi.h>

#define SIZE (65536)
#define FILENAME "iodata"

main(argc, argv)

     int argc;
     char **argv;

{
        int *buf, i, rank, nints, len, flag;
        char *filename;
        MPI_File fh;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        buf = (int *) malloc(SIZE);
        nints = SIZE/sizeof(int);
        for (i=0; i<nints; i++) buf[i] = rank*100000 + i;

        /* each process opens a separate file called FILENAME.'myrank' */

        filename = (char *) malloc(strlen(FILENAME) + 10);
        sprintf(filename, "%s.%d", FILENAME, rank);

        MPI_File_open(MPI_COMM_SELF, filename,
                      MPI_MODE_CREATE | MPI_MODE_RDWR,
                      MPI_INFO_NULL, &fh);

        MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT, "native",
                          MPI_INFO_NULL);
        MPI_File_write(fh, buf, nints, MPI_INT, &status);
        MPI_File_close(&fh);

        /* reopen the file and read the data back */

        for (i=0; i<nints; i++) buf[i] = 0;
        MPI_File_open(MPI_COMM_SELF, filename,
                      MPI_MODE_CREATE | MPI_MODE_RDWR,
                      MPI_INFO_NULL, &fh);
```

```
        MPI_File_set_view(fh, (MPI_Offset)0, MPI_INT, MPI_INT, "native",
                          MPI_INFO_NULL);
        MPI_File_read(fh, buf, nints, MPI_INT, &status);
        MPI_File_close(&fh);

        /* check if the data read is correct */
        flag = 0;
        for (i=0; i<nints; i++)
                if (buf[i] != (rank*100000 + i)) {
                        printf("Process %d: error, read %d, should be %d\n",
                               rank, buf[i], rank*100000+i);
                        flag = 1;
                }

        if (!flag) {
                printf("Process %d: data read back is correct\n", rank);
                MPI_File_delete(filename, MPI_INFO_NULL);
        }

        free(buf);
        free(filename);

        MPI_Finalize();
        exit(0);
}
```

io output

The output from running the io executable is shown below. The application was run with `-np = 4`.

```
Process 1: data read back is correct
Process 3: data read back is correct
Process 2: data read back is correct
Process 0: data read back is correct
```

thread_safe.c

In this C example, N clients loop MAX_WORK times. As part of a single work item, a client must request service from one of Nservers at random. Each server keeps a count of the requests handled and prints a log of the requests to stdout.

```
#include <stdio.h>
#include <mpi.h>
#include <pthread.h>

#define MAX_WORK        40
#define SERVER_TAG      88
#define CLIENT_TAG      99
#define REQ_SHUTDOWN    -1

static int service_cnt = 0;

int process_request(request)
int request;
{
      if (request != REQ_SHUTDOWN) service_cnt++;
      return request;
}

void* server(args)
void *args;

{
        int rank, request;
        MPI_Status status;
        rank = *((int*)args);

        while (1) {
                MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE,
                         SERVER_TAG, MPI_COMM_WORLD, &status);

                if (process_request(request) == REQ_SHUTDOWN)
                      break;

                    MPI_Send(&rank, 1, MPI_INT, status.MPI_SOURCE,
                             CLIENT_TAG, MPI_COMM_WORLD);

                printf("server [%d]: processed request %d for client %d\n",
                       rank, request, status.MPI_SOURCE);
       }

       printf("server [%d]: total service requests: %d\n", rank, service_cnt);
       return (void*) 0;
}
```

```
void client(rank, size)
int rank;
int size;

{
        int w, server, ack;
        MPI_Status status;

        for (w = 0; w < MAX_WORK; w++) {
               server = rand()%size;

        MPI_Sendrecv(&rank, 1, MPI_INT, server, SERVER_TAG, &ack,
                     1,MPI_INT,server,CLIENT_TAG,MPI_COMM_WORLD, &status);

        if (ack != server) {
                   printf("server failed to process my request\n");
                   MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
                }
        }
}

void shutdown_servers(rank)
int rank;

{
         int request_shutdown = REQ_SHUTDOWN;
         MPI_Barrier(MPI_COMM_WORLD);
         MPI_Send(&request_shutdown, 1, MPI_INT, rank, SERVER_TAG, MPI_COMM_WORLD);
}

main(argc, argv)
int argc;
char *argv[];

{
        int rank, size, rtn;
        pthread_t mtid;
        MPI_Status      status;
        int my_value, his_value;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        rtn = pthread_create(&mtid, 0, server, (void*)&rank);
        if (rtn != 0) {
                printf("pthread_create failed\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

        client(rank, size);
        shutdown_servers(rank);
```

```
        rtn = pthread_join(mtid, 0);
        if (rtn != 0) {
                printf("pthread_join failed\n");
                MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

        MPI_Finalize();
        exit(0);
}
```

thread_safe output

The output from running the thread_safe executable is shown below. The application was run with `-np` = 2.

```
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
```

```
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [1]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 1 for client 1
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [0]: processed request 0 for client 0
server [1]: processed request 0 for client 0
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [1]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: processed request 1 for client 1
server [0]: total service requests: 48
server [1]: total service requests: 32
```

B XMPI resource file

This appendix displays the contents of the XMPI Xresource file stored in /opt/mpi/lib/X11/app-defaults/XMPI.

You should make your own copy of the resource file when you wish to customize the contents. Set your Xresource environment in one of the following ways:

- By default, the XMPI utility uses the XMPI Xresource file in /opt/mpi/lib/X11/app-defaults/XMPI. If you move your HP MPI product from it's default /opt/mpi install location, set the MPI_ROOT environment variable to point to the new location. Also set the X application resource environment variable to point to your XMPI resource file. To set the X application resource environment variable enter:

  ```
  % setenv XAPPLRESDIR $MPI_ROOT/lib/X11/
  app-defaults/XMPI
  ```

- You can copy the XMPI resource file to another location and customize it. Set the XAPPLRESDIR environment variable to point to the new XMPI file. For example, if you copy the XMPI file to your home directory, type the following command:

  ```
  % setenv XAPPLRESDIR $HOME/XMPI
  ```

- You can copy the contents of XMPI to the .Xdefaults file in your home directory and customize it. If you change your .Xdefaults file during your login session, you can load the specifications immediately by typing the following command at a shell prompt:

  ```
  % xrdb -load $HOME/.Xdefaults
  ```

The following section displays the contents of the /opt/mpi/lib/X11/app-defaults/XMPI Xresource file:

```
XMPI*Title:XMPI
XMPI*IconName:XMPI
XMPI*multiClickTime:500
XMPI*background:lightgray
XMPI*fontList:-*-helvetica-bold-r-normal--*-120-*-*-*-*-*-*
XMPI*msgFont:-*-helvetica-medium-r-normal--*-120-*-*-*-*-*-*
XMPI*fo_func.fontList:-*-helvetica-bold-o-normal--*-120-*-*-*-*-*
-*
XMPI*dt_dtype.fontList:-*-helvetica-medium-r-normal--*-100-*-*-*-
*-*-*
XMPI*ctl_bar.bottomShadowColor:darkslateblue
XMPI*ctl_bar.background:slateblue
XMPI*ctl_bar.foreground:white
XMPI*banner.background:slateblue
XMPI*banner.foreground:white
XMPI*view_draw.background:black
XMPI*view_draw.foreground:gray
XMPI*trace_draw.foreground:black
XMPI*kiviat_draw.background:gray
XMPI*kiviat_draw.foreground:black
XMPI*matrix_draw.background:gray
XMPI*matrix_draw.foreground:black
XMPI*app_list.visibleItemCount:8
XMPI*aschema_text.columns:24
XMPI*prog_mgr*columns:16
XMPI*comCol:cyan
XMPI*rcomCol:plum
XMPI*label_frame.XmLabel.background:#D3B5B5
XMPI*XmToggleButtonGadget.selectColor:red
XMPI*XmToggleButton.selectColor:red
```

NOTE HP MPI 1.7 is the last release that will support XMPI.

C MPI 2.0 features supported

HP MPI is fully compliant with the MPI 1.2 standard and supports a subset of the MPI 2.0 standard. The MPI 2.0 features supported are identified in Table 11.

Table 11 MPI 2.0 features supported in HP MPI

MPI 2.0 feature	Standard reference
MPI I/O	Chapter 9
Language interoperability	Section 4.12
Thread-compliant library	Section 8.7
MPI_Init NULL arguments	Section 4.2
One-sided communication	Chapter 6
Miscellaneous features	Sections 4.6 through 4.10 and section 8.3

Each of these features is briefly described in the sections of this appendix.

MPI I/O

UNIX I/O functions provide a model for a portable file system. However, the portability and optimization needed for parallel I/O cannot be achieved using this model.

The MPI 2.0 standard defines an interface for parallel I/O that supports partitioning of file data among processes. The standard also supports a collective interface for transferring global data structures between processes memories and files.

HP MPI I/O supports a subset of the MPI 2.0 standard using ROMIO, a portable implementation developed at Argonne National Laboratory. The subset is identified in Table 12.

Table 12 MPI I/O functionality supported by HP MPI

I/O functionality	Standard reference
File manipulation	section 9.2
File views	section 9.3
Data access	section 9.4 except sections 9.4.4 and 9.4.5
Consistency and semantics	section 9.6

HP MPI I/O has the following limitations:

- All nonblocking I/O requests use a MPIO_Request object instead of MPI_Request. The `MPIO_Test` and `MPIO_Wait` routines are provided to test and wait for MPIO_Request objects. `MPIO_Test` and `MPIO_Wait` have the same semantics as `MPI_Test` and `MPI_Wait` respectively.
- The status argument is not returned in any MPI I/O operation.
- All calls that involve MPI I/O file offsets must use an 8-byte integer. Because HP-UX Fortran 77 only supports 4-byte integers, all Fortran 77 source files that involve file offsets must be compiled using HP-UX Fortran 90. In this case, the Fortran 90 offset is defined by

  ```
  integer (KIND=MPI_OFFSET_KIND)
  ```

- Some I/O routines (for example, `MPI_File_open`, `MPI_File_delete`, and `MPI_File_set_info`) take an input argument called *info*. Refer to Table 13 for supported keys for this argument.

Table 13 `Info` object keys

Key	Information
cb_buffer_size	Buffer size for collective I/O
cb_nodes	Number of processes that actually perform I/O in collective I/O
ind_rd_buffer_size	Buffer size for data sieving in independent reads
ind_wr_buffer_size	Buffer size for data sieving in independent writes

NOTE If a given key is not supported or if the value is invalid, the key is ignored.

The example C code, “io.c” on page 156 demonstrates the use of MPI 2.0 standard parallel IO functions. The io.c program has functions to manipulate files, access data, and change the process's view of data in the file.

Language interoperability

Language interoperability allows you to write mixed-language applications or applications that call library routines written in another language. For example, you can write applications in Fortran or C that call MPI library routines written in C or Fortran respectively.

MPI provides a special set of conversion routines for converting objects between languages. You can convert MPI communicators, data types, groups, requests, reduction operations, and status objects. Conversion routines are described in Table 14.

Table 14 Language interoperability conversion routines

Routine	Description
`MPI_Fint MPI_Comm_c2f(MPI_Comm);`	Converts a C communicator handle into a Fortran handle
`MPI_Comm MPI_Comm_f2c(MPI_Fint);`	Converts a Fortran communicator handle into a C handle
`MPI_Fint MPI_Type_c2f(MPI_Datatype);`	Converts a C data type into a Fortran data type
`MPI_Datatype MPI_Type_f2c(MPI_Fint);`	Converts a Fortran data type into a C data type
`MPI_Fint MPI_Group_c2f(MPI_Group);`	Converts a C group into a Fortran group
`MPI_Group MPI_Group_f2c(MPI_Fint);`	Converts a Fortran group into a C group
`MPI_Fint MPI_Op_c2f(MPI_Op);`	Converts a C reduction operation into a Fortran reduction operation
`MPI_Op MPI_Op_f2c(MPI_Fint);`	Converts a Fortran reduction operation into a C reduction operation
`MPI_Fint MPI_Request_c2f(MPI_Request);`	Converts a C request into a Fortran request

Routine	Description
`MPI_Request MPI_Request_f2c(MPI_Fint);`	Converts a Fortran request into a C request
`int MPI_Status_c2f(MPI_Status *, MPI_Fint *);`	Converts a C status into a Fortran status
`int MPI_Status_f2c(MPI_Fint *, MPI_Status *);`	Converts a Fortran status into a C status
`MPI_file MPI_File_f2c (MPI_Fint file)`	Converts a Fortran file handle into a C file handle
`MPI_Fint MPI_File_c2f (MPI_File file)`	Converts a C file handle into a Fortran file handle

Thread-compliant library

HP MPI provides a thread-compliant library for applications running under HP-UX 11.0 (32-and 64-bits). On HP-UX 11.0, HP MPI supports concurrent MPI calls by threads, and a blocking MPI call blocks only the invoking thread, allowing another thread to be scheduled.

By default, the non thread-compliant library (libmpi) is used when running MPI jobs. Linking to the thread-compliant library (libmtmpi) is now required only for applications that have multiple threads making MPI calls simultaneously. Table 15 shows which library to use for a given HP MPI application type.

Table 15 HP MPI library usage

Application type	Library to link		Comments
Non-threaded MPI application		libmpi	Most MPI applications
Non-threaded MPI application with mostly nonblocking communication	libmtmpi	libmpi	Potential performance improvement if run with libmtmpi and the communication thread (`MPI_MT_FLAGS=ct`)
Non-parallel MLIB applications (link with `-lveclib`)		libmpi	
Thread parallel MLIB applications (link with `-lveclib`)		libmpi	
+O3 +Oparallel		libmpi	
Using pthreads	libmtmpi	libmpi	If the user is explicitly using pthreads, and they guarantee that no 2 threads call MPI at the same time, libmpi can be used. Otherwise, use libmtmpi.
libmpi represents the non thread-compliant library *libmtmpi represents the thread-compliant library*			

NOTE When you use the thread-compliant library, overall performance is a function of the level of thread support required by the application. Thread support levels are described in Table 16 on page 172.

To link with the thread-compliant library, use the `-libmtmpi` option when compiling your application.

To create a communication thread for each process in your job (for example, to overlap computation and communication), specify the `ct` option in the MPI_MT_FLAGS environment variable. See "MPI_MT_FLAGS" on page 44 for more information.

Alternatively, you may set the `s[a][p]#` option for the MPI_FLAGS environment variable. For the thread-compliant library, setting `MPI_FLAGS=s[a][p]#` has the same effect as setting `MPI_MT_FLAGS=ct` when the value of # is greater than 0. `MPI_MT_FLAGS=ct` takes priority over the default `MPI_FLAGS=sp0` setting. Refer to "MPI_FLAGS" on page 37.

To set the level of thread support for your job, you can specify the appropriate run time option in `MPI_MT_FLAGS` or modify your application to use `MPI_Init_thread` instead of `MPI_Init`.

To modify your application, replace the call to `MPI_Init` with

MPI_Init_thread(int *argc, char *((*argv) []), int *required*, int **provided*));

where

required Specifies the desired level of thread support.

provided Specifies the provided level of thread support.

Table 16 shows the possible thread-initialization values for *required* and the values returned by *provided* for the non thread-compliant library (libmpi) and for the thread-compliant library (libmtmpi).

Table 16 Thread-initialization values

MPI library	Value for *required*	Value returned by *provided*
libmpi	MPI_THREAD_SINGLE	MPI_THREAD_SINGLE
libmpi	MPI_THREAD_FUNNELED	MPI_THREAD_SINGLE
libmpi	MPI_THREAD_SERIALIZED	MPI_THREAD_SINGLE
libmpi	MPI_THREAD_MULTIPLE	MPI_THREAD_SINGLE
libmtmpi	MPI_THREAD_SINGLE	MPI_THREAD_SINGLE
libmtmpi	MPI_THREAD_FUNNELED	MPI_THREAD_FUNNELED
libmtmpi	MPI_THREAD_SERIALIZED	MPI_THREAD_SERIALIZED
libmtmpi	MPI_THREAD_MULTIPLE	MPI_THREAD_MULTIPLE

Table 17 shows the relationship between the possible thread-support levels in `MPI_Init_thread` and the corresponding options in `MPI_MT_FLAGS`.

Table 17 Thread-support levels

MPI_Init_thread	MPI_MT_FLAGS	Behavior
MPI_THREAD_SINGLE	single	Only one thread will execute
MPI_THREAD_FUNNELED	fun	The process may be multithreaded, but only the main thread will make MPI calls
MPI_THREAD_SERIALIZED	serial	The process may be multithreaded, and multiple threads can make MPI calls, but only one call can be made at a time
MPI_THREAD_MULTIPLE	mult	Multiple threads may call MPI at any time with no restrictions. This option is the default

Refer to example “thread_safe.c” on page 158 for a program that uses multiple threads.

To prevent application deadlock, do not call the thread-compliant library from a signal handler or cancel a thread that is executing inside an MPI routine.

Counter instrumentation (refer to “Using counter instrumentation” on page 68) is supported for the thread-compliant library in addition to the standard MPI library. Therefore you can collect profiling information for applications linked with the thread-compliant library.

The thread-compliant library supports calls to the following MPI 2.0 standard functions:

- `MPI_Init_thread`
- `MPI_Is_thread_main`
- `MPI_Query_thread`

No other MPI 2.0 calls are supported in the thread-compliant library.

MPI_Init NULL arguments

In MPI-1.1, it is explicitly stated that an implementation is allowed to require that the arguments argc and argv passed by an application to MPI_INIT in C be the same arguments passed into the application as the arguments to main. In MPI-2, implementations are not allowed to impose this requirement.

HP MPI complies with this MPI-2 standard extension by allowing applications to pass NULL for both the argc and argv arguments of main. However, MPI_Init (NULL, NULL) is supported only when you use mpirun to run your MPI application. For example, use one of the following:

```
% mpirun -np 4 my_program
```

```
% mpirun -f my_appfile
```

Refer to “Compiling and running your first application” on page 19 and “mpirun” on page 49 for details about the methods to run your HP MPI application.

One-sided communication

Message-passing communication involves transferring data from the sending process to the receiving process and synchronization of the sender and receiver.

Remote memory access and one-sided communication extend the communication mechanism of MPI by separating the communication and synchronization functions. One process specifies all communication parameters both for the sending side and the receiving side. This mode of communication is best for applications with dynamically changing data access patterns where data distribution is fixed or slowly changing. Each process can compute what data it needs to access or update at other processes. Processes in such applications, however, may not know which data in their memory needs to be accessible by remote processes or even the identity of these remote processes.

In this case, applications can open windows in their memory space that are accessible by remote processes.

HP MPI supports a subset of the MPI 2.0 one-sided communication functionality:

- Window creation—The initialization process that allows each process in an intracommunicator group to specify, in a collective operation, a window in its memory that is made accessible to remote processes. The window-creation call returns an opaque object that represents the group of processes that own and access a set of windows, and the attributes of each window, as specified by the initialization call.

  HP MPI supports the `MPI_Win_create` and the `MPI_Win_free` functions. `MPI_Win_Create` is a collective call executed by all processes in a group. It returns a window object that can be used by these processes to perform remote memory access operations. `MPI_Win_free` is also a collective call, and frees the window object created by `MPI_Win_create`, and returns a null handle.

- Window attributes—HP MPI supports the `MPI_Win_get_group` function. `MPI_Win_get_group` returns a duplicate of the group of the communicator used to create the window, that is, the processes that share access to the window.

- Data transfer—Data transfer operations are nonblocking: data transfer calls initiate the transfer, but the transfer may continue after the call returns. The transfer is completed, both at the origin and at the target, when a subsequent synchronization call is issued by the caller on the involved window object.

  HP MPI supports two data transfer operations; `MPI_Put` and `MPI_Get`. `MPI_Put` is similar to execution of a send by the origin process and a matching receive by the target process, except that all arguments are provided by the call executed by the origin process.

- Synchronization—Transfer operations complete at the origin and at the target, when a subsequent synchronization call is issued by the caller on the involved window object.

  HP MPI supports three synchronization calls; `MPI_Win_fence`, `MPI_Win_lock`, and `MPI_Win_unlock`.

  `MPI_Win_fence` is a collective synchronization call that supports a loosely synchronous model, where global computation phases alternate with global communication phases. All remote memory access calls originating at a given process, and started before the fence call, complete at that process before the fence call returns. Remote memory access operations started by a process after the fence call returns, access their target window only after `MPI_Win_fence` has been called by the target process.

  `MPI_Win_lock` and `MPI_Win_unlock` start and complete a remote memory access epoch, respectively. Remote memory access operations issued during the epoch complete at the origin and at the target before `MPI_Win_unlock` returns.

Restrictions for the HP MPI implementation of one-sided communication include:

- MPI window segments must be allocated using `MPI_Alloc_mem`; they cannot be placed in COMMON blocks, the stack, or the heap.
- Multi-host user programs that call one-sided communication functions must be started by `mpirun` with the `-commd` option. This option is not required on single-host programs.
- `MPI_Accumulate` is not supported.
- Non-contiguous derived data types are not supported for one-sided communications.
- One-sided communications are not supported in the diagnostic library.
- One-sided communications are not supported in the multithreaded library

Miscellaneous features

Miscellaneous features supported from sections 4.6 through 4.10 and section 8.3 through 8.4 of the MPI 2.0 standard include:

- Committing a committed datatype—Allows `MPI_Type_commit` to accept committed datatypes. In this case, no action is taken.
- Allowing user functions at process termination—Defines what actions take place when a process terminates. These actions are specified by attaching an attribute to `MPI_COMM_SELF` with a callback function. When `MPI_FINALIZE` is called, it first executes the equivalent of an `MPI_COMM_FREE` on `MPI_COMM_SELF`. This causes the delete callback function to be called on all keys associated with `MPI_COMM_SELF`. The freeing of `MPI_COMM_SELF` occurs before any other part of MPI is affected.
- Determining whether MPI has finished—Allows layered libraries to determine whether MPI is still active by using `MPI_Finalize`.
- Using the `Info` object—Provides system-dependent hints. Sets *key* and *value* pairs (both *key* and *value* are strings) for the opaque information object, `Info`. `Info` object routines include those described in Table 18 on page 179.
- Associating information with status—Sets the number of elements to associate with the status for requests. In addition, sets the status to associate with the cancel flag to indicate whether a request was cancelled. Status routines include:

`MPI_Status_set_elements`	Modifies the opaque part of status.
`MPI_Status_set_cancelled`	Indicates whether a status request is cancelled.

- Associate a name with a communicator, a window, or a datatype—Allows you to associate a printable identifier with an HP MPI communicator, window, or datatype. This can be useful for error reporting, debugging, and profiling. Routines used to associate names with objects include those described in Table 19 on page 179.

Table 18 **`Info` object routines**

Object routine	Function
MPI_Info_create	Creates a new `info` object
MPI_Info_set	Adds the (key, value) pair to `info` and overrides the value if a value for the same key was previously set
MPI_Info_delete	Deletes a (key, value) pair from `info`
MPI_Info_get	Retrieves the value associated with key in a previous call to `MPI_Info_set`
MPI_Info_get_valuelen	Retrieves length of the value associated with key
MPI_Info_get_nkeys	Returns the number of keys currently defined in `info`
MPI_Info_get_nthkey	Returns the *nth* defined key in `info`
MPI_Info_dup	Duplicates an existing `info` object, creating a new object with the same (key, value) pairs and ordering of keys
MPI_Info_free	Frees the `info` object

Table 19 **Naming object routines**

Object routine	Function
MPI_Comm_set_name	Associates a name string with a communicator
MPI_Comm_get_name	Returns the last name that was associated with a given communicator
MPI_Type_set_name	Associates a name string with a datatype
MPI_Type_get_name	Returns the last name that was associated with a given datatype
MPI_Win_set_name	Associates a name string with a window
MPI_Win_get_name	Returns the last name that was associated with a given window

D Standard-flexibility in HP MPI

HP MPI is fully compliant with the MPI 1.2 standard and supports the subset of the MPI 2.0 standard described in Appendix C, “MPI 2.0 features supported” . There are items in the MPI standard for which the standard allows flexibility in implementation. This appendix identifies HP MPI's implementation of many of these standard-flexible issues.

Table 20 displays references to sections in the MPI standard that identify flexibility in the implementation of an issue. Accompanying each reference is HP MPI's implementation of that issue.

Table 20 HP MPI implementation of standard-flexible issues

Reference in MPI standard	HP MPI's implementation
MPI implementations are required to define the behavior of MPI_Abort (at least for a `comm` of `MPI_COMM_WORLD`). MPI implementations may ignore the `comm` argument and act as if `comm` was `MPI_COMM_WORLD`. *See MPI-1.2 Section* 7.5.	MPI_Abort kills the application. `comm` is ignored, uses `MPI_COMM_WORLD`.
An implementation must document the implementation of different language bindings of the MPI interface if they are layered on top of each other. *See MPI-1.2 Section* 8.1.	Fortran is layered on top of C and profile entry points are given for both languages.
MPI does not mandate what an MPI process is. MPI does not specify the execution model for each process; a process can be sequential or multithreaded. *See MPI-1.2 Section* 2.6.	MPI processes are UNIX processes and can be multithreaded.
MPI does not provide mechanisms to specify the initial allocation of processes to an MPI computation and their initial binding to physical processes. *See MPI-1.2 Section* 2.6.	HP MPI provides the `mpirun -np #` utility and appfiles. Refer to the relevant sections in this guide.
MPI does not mandate that any I/O service be provided, but does suggest behavior to ensure portability if it is provided. *See MPI-1.2 Section* 2.8.	Each process in HP MPI applications can read and write data to an external drive. Refer to “External input and output” on page 126 for details.

Reference in MPI standard (Continued)	HP MPI's implementation (Continued)
The value returned for `MPI_HOST` gets the rank of the host process in the group associated with `MPI_COMM_WORLD`. `MPI_PROC_NULL` is returned if there is no host. MPI does not specify what it means for a process to be a host, nor does it specify that a `HOST` exists.	HP MPI always sets the value of `MPI_HOST` to `MPI_PROC_NULL`.
MPI provides `MPI_GET_PROCESSOR_NAME` to return the name of the processor on which it was called at the moment of the call. *See MPI-1.2 Section* 7.1.1.	If you do not specify a host name to use, the hostname returned is that of the UNIX gethostname(2). If you specify a host name using the `-h` option to `mpirun`, HP MPI returns that host name.
The current MPI definition does not require messages to carry data type information. Type information might be added to messages to allow the system to detect mismatches. *See MPI-1.2 Section* 3.3.2.	The default HP MPI library does not carry this information due to overload, but the HP MPI diagnostic library (DLIB) does. To link with the diagnostic library, use `-ldmpi` on the link line.
Vendors may write optimized collective routines matched to their architectures or a complete library of collective communication routines can be written using MPI point-to-point routines and a few auxiliary functions. *See MPI-1.2 Section* 4.1.	Use HP MPI's collective routines instead of implementing your own with point-to-point routines. HP MPI's collective routines are optimized to use shared memory where possible for performance.
Error handlers in MPI take as arguments the communicator in use and the error code to be returned by the MPI routine that raised the error. An error handler can also take "stdargs" arguments whose number and meaning is implementation dependent. *See MPI-1.2 Section* 7.2 and *MPI-2.0 Section* 4.12.6.	To ensure portability, HP MPI's implementation does not take "stdargs". For example in C, the user routine should be a C function of type `MPI_handler_function`, defined as: void (MPI_Handler_function) (MPI_Comm *, int *);

Reference in MPI standard (Continued)	HP MPI's implementation (Continued)
MPI implementors may place a barrier inside MPI_FINALIZE. *See MPI-2.0 Section* 3.2.2.	HP MPI's `MPI_FINALIZE` behaves as a barrier function such that the return from `MPI_FINALIZE` is delayed until all potential future cancellations are processed.
MPI defines minimal requirements for thread-compliant MPI implementations and MPI can be implemented in environments where threads are not supported. *See MPI-2.0 Section* 8.7.	HP MPI provides a thread-compliant library (libmtmpi). Use `-libmtmpi` on the link line to use the libmtmpi. Refer to "Thread-compliant library" on page 170 for more information.
The format for specifying the filename in `MPI_FILE_OPEN` is implementation dependent. An implementation may require that filename include a string specifying additional information about the file. *See MPI-2.0 Section* 9.2.1.	HP MPI I/O supports a subset of the MPI 2.0 standard using ROMIO, a portable implementation developed at Argonne National Laboratory. No additional file information is necessary in your filename string.

Glossary

asynchronous Communication in which sending and receiving processes place no constraints on each other in terms of completion. The communication operation between the two processes may also overlap with computation.

bandwidth Reciprocal of the time needed to transfer a byte. Bandwidth is normally expressed in megabytes per second.

barrier Collective operation used to synchronize the execution of processes. `MPI_Barrier` blocks the calling process until all receiving processes have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

blocking receive Communication in which the receiving process does not return until its data buffer contains the data transferred by the sending process.

blocking send Communication in which the sending process does not return until its associated data buffer is available for reuse. The data transferred can be copied directly into the matching receive buffer or a temporary system buffer.

broadcast One-to-many collective operation where the root process sends a message to all other processes in the communicator including itself.

buffered send mode Form of blocking send where the sending process returns when the message is buffered in application-supplied space or when the message is received.

buffering Amount or act of copying that a system uses to avoid deadlocks. A large amount of buffering can adversely affect performance and make MPI applications less portable and predictable.

cluster Group of computers linked together with an interconnect and software that functions collectively as a parallel machine.

collective communication Communication that involves sending or receiving messages among a group of processes at the same time. The communication can be one-to-many, many-to-one, or many-to-many. The main collective routines are `MPI_Bcast`, `MPI_Gather`, and `MPI_Scatter`.

communicator Global object that groups application processes together. Processes in a communicator can communicate with each other or with processes in another group. Conceptually, communicators define a communication context and a static group of processes within that context.

context Internal abstraction used to define a safe communication space for processes. Within a communicator, context separates point-to-point and collective communications.

data-parallel model Design model where data is partitioned and distributed to each process in an application. Operations are performed on each set of data in parallel and intermediate results are exchanged between processes until a problem is solved.

derived data types User-defined structures that specify a sequence of basic data types and integer displacements for noncontiguous data. You create derived data types through the use of type-constructor functions that describe the layout of sets of primitive types in memory. Derived types may contain arrays as well as combinations of other primitive data types.

determinism A behavior describing repeatability in observed parameters. The order of a set of events does not vary from run to run.

domain decomposition Breaking down an MPI application's computational space into regular data structures such that all computation on these structures is identical and performed in parallel.

explicit parallelism Programming style that requires you to specify parallel constructs directly. Using the MPI library is an example of explicit parallelism.

functional decomposition Breaking down an MPI application's computational space into separate tasks such that all computation on these tasks is performed in parallel.

gather Many-to-one collective operation where each process (including the root) sends the contents of its send buffer to the root.

granularity Measure of the work done between synchronization points. Fine-grained applications focus on execution at the instruction level of a program. Such applications are load balanced but suffer from a low computation/communication ratio. Coarse-grained applications focus on execution at the program level where multiple programs may be executed in parallel.

group Set of tasks that can be used to organize MPI applications. Multiple groups are useful for solving problems in linear algebra and domain decomposition.

implicit parallelism Programming style where parallelism is achieved by software layering (that is, parallel constructs are generated through the software). High performance Fortran is an example of implicit parallelism.

intercommunicators Communicators that allow only processes within the same group or in two different groups to exchange data. These communicators support only point-to-point communication.

intracommunicators Communicators that allow processes within the same group to exchange data. These communicators support both point-to-point and collective communication.

instrumentation Cumulative statistical information collected and stored in ascii format. Instrumentation is the recommended method for collecting profiling data.

latency Time between the initiation of the data transfer in the sending process and the arrival of the first byte in the receiving process.

load balancing Measure of how evenly the work load is distributed among an application's processes. When an application is perfectly balanced, all processes share the total work load and complete at the same time.

locality Degree to which computations performed by a processor depend only upon local data. Locality is measured in several ways including the ratio of local to nonlocal data accesses.

message bin A message bin stores messages according to message length. You can define a message bin by defining the byte range of the message to be stored in the bin—use the MPI_INSTR environment variable.

message-passing model Model in which processes communicate with each other by sending and receiving messages. Applications based on message passing are nondeterministic by default. However, when one process sends two or more messages to another, the transfer is deterministic as the messages are always received in the order sent.

MIMD Multiple instruction multiple data. Category of applications in which many instruction streams are applied concurrently to multiple data sets.

MPI Message-passing interface. Set of library routines used to design scalable parallel applications. These routines provide a wide range of operations that include computation, communication, and synchronization. MPI 1.2 is the current standard supported by major vendors.

MPIVIEW An HP MPI utility that is a graphical user interface to display instrumentation data collected at run time.

MPMD Multiple data multiple program. Implementations of HP MPI that use two or more separate executables to construct an application. This design style can be used to simplify the application source and reduce the size of spawned processes. Each process may run a different executable.

multilevel parallelism Refers to multithreaded processes that call MPI routines to perform computations. This approach is beneficial for problems that can be decomposed into logical parts for parallel execution (for example, a looping construct that spawns multiple threads to perform a computation and then joins after the computation is complete).

multihost A mode of operation for an MPI application where a cluster is used to carry out a parallel application run.

nonblocking receive Communication in which the receiving process returns before a message is stored in the receive buffer. Nonblocking receives are useful when communication and computation can be effectively overlapped in an MPI application. Use of nonblocking receives may also avoid system buffering and memory-to-memory copying.

nonblocking send Communication in which the sending process returns before a message is stored in the send buffer. Nonblocking sends are useful when communication and computation can be effectively overlapped in an MPI application.

non–determinism A behavior describing non repeatable observed parameters. The order of a set of events depends on run time conditions and so varies from run to run.

parallel efficiency An increase in speed in the execution of a parallel application.

point-to-point communication Communication where data transfer involves sending and receiving messages between two processes. This is the simplest form of data transfer in a message-passing model.

polling Mechanism to handle asynchronous events by actively checking to determine if an event has occurred.

process Address space together with a program counter, a set of registers, and a stack. Processes can be single threaded or multithreaded. Single-threaded processes can only perform one task at a time. Multithreaded processes can perform multiple tasks concurrently as when overlapping computation and communication.

race condition Situation in which multiple processes vie for the same resource and receive it in an unpredictable manner. Race conditions can lead to cases where applications do not run correctly from one invocation to the next.

rank Integer between zero and (number of processes - 1) that defines the order of a process in a communicator. Determining the rank of a process is important when solving problems where a master process partitions and distributes work to slave processes. The slaves perform some computation and return the result to the master as the solution.

ready send mode Form of blocking send where the sending process cannot start until a matching receive is posted. The sending process returns immediately.

reduction Binary operations (such as summation, multiplication, and boolean) applied globally to all processes in a communicator. These operations are only valid on numeric data and are always associative but may or may not be commutative.

scalable Ability to deliver an increase in application performance proportional to an increase in hardware resources (normally, adding more processors).

scatter One-to-many operation where the root's send buffer is partitioned into *n* segments and distributed to all processes such that the *i*th process receives the *i*th segment. *n* represents the total number of processes in the communicator.

send modes Point-to-point communication in which messages are passed using one of four different types of blocking sends. The four send modes include standard mode (`MPI_Send`), buffered mode (`MPI_Bsend`), synchronous mode (`MPI_Ssend`), and ready mode (`MPI_Rsend`). The modes are all invoked in a similar manner and all pass the same arguments.

shared memory model Model in which each process can access a shared address space. Concurrent accesses to shared memory are controlled by synchronization primitives.

SIMD Single instruction multiple data. Category of applications in which homogeneous processes execute the same instructions on their own data.

SMP Symmetric multiprocessor. A multiprocess computer in which all the processors have equal access to all machine resources. Symmetric multiprocessors have no manager or worker processes.

spin-yield Refers to an HP MPI facility that allows you to specify the number of milliseconds a process should block (spin) waiting for a message before yielding the CPU to another

process. Specify a spin-yield value in the MPI_FLAGS environment variable.

SPMD Single program multiple data. Implementations of HP MPI where an application is completely contained in a single executable. SPMD applications begin with the invocation of a single process called the master. The master then spawns some number of identical child processes. The master and the children all run the same executable.

standard send mode Form of blocking send where the sending process returns when the system can buffer the message or when the message is received.

stride Constant amount of memory space between data elements where the elements are stored noncontiguously. Strided data are sent and received using derived data types.

synchronization Bringing multiple processes to the same point in their execution before any can continue. For example, `MPI_Barrier` is a collective routine that blocks the calling process until all receiving processes have called it. This is a useful approach for separating two stages of a computation so messages from each stage are not overlapped.

synchronous send mode Form of blocking send where the sending process returns only if a matching receive is posted and the receiving process has started to receive the message.

tag Integer label assigned to a message when it is sent. Message tags are one of the synchronization variables used to ensure that a message is delivered to the correct receiving process.

task Uniquely addressable thread of execution.

thread Smallest notion of execution in a process. All MPI processes have one or more threads. Multithreaded processes have one address space but each process thread contains its own counter, registers, and stack. This allows rapid context switching because threads require little or no memory management.

thread-compliant An implementation where an MPI process may be multithreaded. If it is, each thread can issue MPI calls. However, the threads themselves are not separately addressable.

trace Information collected during program execution that you can use to analyze your application. You can collect trace information and store it in a file for later use or analyze it directly when running your application interactively (for example, when you run an application in the XMPI utility).

yield See spin-yield.

XMPI An X/Motif graphical user interface for running applications, monitoring processes and messages, and viewing trace files.

Index

Symbols
+autodbl, 29
+autodbl4, 29
+autodblpad, 29
+DA2 option, 29
+DD64 option, 29
.mpiview file, 68
.tr file, 79
.tr output file, 92
/opt/aCC/bin/aCC, 28
/opt/ansic/bin/cc, 28
/opt/fortran/bin/f77, 28
/opt/fortran90/bin/f90, 28
/opt/mpi
  subdirectories, 25
/opt/mpi directory
  organization of, 25
/opt/mpi/bin, 25
/opt/mpi/doc/html, 25
/opt/mpi/help, 25
/opt/mpi/include, 25
/opt/mpi/lib/hpux32, 25
/opt/mpi/lib/hpux64, 25
/opt/mpi/lib/pa1.1/libfmpi.a, 25
/opt/mpi/lib/pa20.64/libfmpi.a, 25
/opt/mpi/lib/X11/app-defaults, 25
/opt/mpi/newconfig/, 25
/opt/mpi/share/man/man1.Z, 25
/opt/mpi/share/man/man3.Z, 25

Numerics
64-bit support, 29

A
abort HP MPI, 98
aCC, 28
ADI *See* alternating direction iterative method
allgather, 10
allows, 104
all-reduce, 12
alltoall, 10
alternating direction iterative method, 131, 147
amount variable, 41
appfile
  configure for multiple network interfaces, 107
  description of, 22
  XMPI interactive mode, 90
appfiles
  adding program arguments, 56
  assigning ranks in, 57
  creating, 55
  improving communication on multihost systems, 57
  setting remote environment variables in, 57
application hangs
  See zero-buffering
argument checking, disable, 40
array partitioning, 148
ASCII instrumentation profile, 69
asynchronous communication, 3
automatic snapshot, 91
Automatic snapshot field, 95

B
backtrace, 119
backward compatibility, 49
bandwidth, 5, 105, 110
barrier, 14, 111
blocked process, 83
blocking communication, 6
  buffered mode, 7
  MPI_Bsend, 7
  MPI_Recv, 8
  MPI_Rsend, 7

MPI_Send, 7
MPI_Ssend, 7
point-to-point, 83
read mode, 7
receive mode, 7, 8
send mode, 7
standard mode, 7
synchronous mode, 7
blocking receive, 8
blocking send, 7
broadcast, 10, 11
buf variable, 7, 8, 9, 11
Buffer size field, 99
buffered send mode, 7
build
examples, 132
MPI on multiple hosts, 21, 55–61
MPI on single host, 20
problems, 122

C

C compiler, 28
C examples
communicator.c, 131, 146
io.c, 156
ping_pong.c, 131, 135
thread_safe.c, 158
C++ compiler, 28
C++ examples
cart.C, 131, 142
cart.C, 131
change
default settings, 95
execution location, 46
viewing options, 95
cnt
See data element count
cnt field, 87, 88
code a
blocking receive, 8
blocking send, 7
broadcast, 11
nonblocking send, 9
scatter, 12
code error conditions, 128
collect profile information
ASCII report, 69
mpiview, 73–76
XMPI interactive mode, 90–99
XMPI postmortem mode, 78–90
See MPIHP_Trace_off
See MPIHP_Trace_on
collective communication, 10, 84
all-reduce, 12
reduce, 12
reduce-scatter, 12
scan, 12
collective operations, 10, 10–14
communication, 10
computation, 12
synchronization, 13
comm field, 87, 88
comm variable, 8, 9, 11, 12, 13
communication
context, 8, 13
hot spot, 70
hot spots, 57
improving interhost, 57
one-sided, 175
using daemons, 62
communicator
defaults, 5
determine no. of processes, 6
freeing memory, 37
communicator.c, 131
commutative reductions, 111
compatibility, 49
compilation
utilities, 26
compiler options
+autodbl, 29
+autodbl4, 29
+autodblpad, 29
+DA2.0W, 29

+DD64, 29
32- and 64-bit library, 29
Fortran, 29
-L, 28
-l, 28
-notv, 28
-Wl, 28
compiling applications, 28
completing HP MPI, 128
completion routine, 7
complying with MPI 2.0 standard
committed datatypes, 178
I/O, 166
Info objects, 178
language interoperability, 168
`MPI_Finalize`, 178
MPI_Init(NULL, NULL), 174
one-sided communication, 175
process termination, 178
status information, 178
thread-compliant library, 170
computation, 12
compute_pi.f, 68, 131
configuration files, 25
configure environment, 18
setenv MPI_ROOT, 25
setenv NLSPATH, 65
setenv XAPPLRESDIR, 163
constructor functions
contiguous, 15
indexed, 15
structure, 15
vector, 15
context
communication, 8, 13
context switching, 109
contiguous and noncontiguous data, 14
contiguous constructor, 15
convert objects between languages, 168
copy field, 88
copy *See* number of message copies sent
corresponding MPI blocking/nonblocking calls, 9
count variable, 7, 8, 9, 11
counter instrumentation, 41, 68
ASCII format, 69
create profile, 68
using mpiview, 73, 77
create
appfile, 55
ASCII profile, 68
instrumentation profile, 68
trace file, 79
CXperf, 100

D

daemons
multipurpose, 58
number of processes, 58
daemons, communication, 62
data element count, 88
DDE, 37, 114, 130
debug HP MPI, 37, 114, 130
See also diagnostic library
See also enhanced debugging output
See also MPI_Flags
debuggers, 114
decrease trace magnification, 83
derived data types, 14
dest variable, 8, 9
determine
group size, 4
no. of messages sent
no. of processes in communicator, 6
rank of calling process, 4

diagnostics library
message signature analysis, 118
MPI object-space corruption, 118
multiple buffer writes detection, 118
using, 118
dial time, 82, 90
dialogs
Kiviat, 84, 85, 89
mpirun options, 97
XMPI Application Browser, 90
XMPI buffer size, 96
XMPI Confirmation, 93
XMPI Datatype, 87
XMPI Express, 93
XMPI Focus, 86
XMPI Kiviat, 89
XMPI monitor options, 95
XMPI Trace, 82, 90
XMPI Trace Selection, 81
directory structure, MPI, 25
distribute sections/compute in parallel, 131, 140
dtype variable, 8, 9, 11, 13
Dump, 92
dump shmem configuration, 40

E

edde, 37, 114, 130
egdb, 37, 114, 130
enable
instrumentation, 23, 50
trace generation, 24, 91, 92, 97
verbose mode, 97
enhanced debugging output, 119
environment variables
MP_GANG, 40
MPI_CC, 29
MPI_COMMD, 35
MPI_CXX, 29
MPI_DLIB_FLAGS, 35
MPI_F77, 29
MPI_F90, 29
MPI_FLAGS, 37, 114
MPI_GLOBMEMSIZE, 41
MPI_INSTR, 41, 68
MPI_LOCALIP, 43
MPI_MT_FLAGS, 44, 45, 46
MPI_REMSH, 45
MPI_SHMEMCNTL, 46
MPI_WORKDIR, 46
MPI_XMPI, 47
NLSPATH, 65
runtime, 34–40
setting in appfiles, 57
TOTALVIEW, 48
XAPPLRESDIR, 163
error checking, disable, 40
error conditions, 128
ewdb, 37, 114, 130
example applications, 131–161
cart.C, 131, 142
communicator.c, 131, 146
compiling and running, 132
compute_pi.f, 68, 131, 138
copy default communicator, 131, 146
distribute sections/compute in parallel, 131, 140
generate virtual topology, 131
io.c, 156
master_worker.f90, 131, 140
measure send/receive time, 131
multi_par.f, 131, 147

ping_pong.c, 131, 135
receive operation, 131
send operation, 131
send_receive.f, 133
thread_safe.c, 158
use ADI on 2D compute region, 131
exceeding file descriptor limit, 126
exdb, 37, 114, 130
Express option
get full trace, 93
get partial trace, 93
See Trace dialog
external input and output, 126

F
FAQ, 129
Fast Forward
See trace file
fast forward trace log, 83
file data partitioning
See I/O, 166
file descriptor limit, 126
Fortran 77 compiler, 28
Fortran 77 examples
array partitioning, 148
compute_pi.f, 131, 138
multi_par.f, 131, 147
send_receive.f, 131, 133
Fortran 90 compiler, 28
Fortran 90 examples
master_worker.f90, 140
Fortran 90 troubleshooting, 125
Fortran compiler options, 29
Fortran profiling, 102
freeing memory, 37
frequently asked questions, 129
full trace, 93
fully subscribed
See subscription types

G
gang scheduling, 40, 109
gather, 10
GDB, 37, 114, 130
gethostname, 181
getting started, 17
ght, 68
global reduce-scatter, 12
global reduction, 12
global variables
MPI_DEBUG_CONT, 114
graph
MPIVIEW, 75
rotate, 76
view multiple, 76
window, 73
zoom, 76
graph legend, 76
green
See process colors
group membership, 3
group size, 4

H
header files, 25
heart-beat signals, 38
hexagons, 90
hosts
assigning using LSF, 64
multiple, 55–61
HP MPI
abort, 98
building, 122
change behavior, 37, 130
clean-up, 129
completing, 128
debug, 113
FAQ, 113, 129
frequently asked questions, 129
jobs running, 59
kill, 61
multi-process debuggers, 116
profile process, 100
running, 123

single-process debuggers, 114
specify shared memory, 41
starting, 50, 122
troubleshooting, 121–130
twisted-data layout, 149
utility files, 25
HP MPI User's Guide (html), 25
HP MPI utility files, 25
HP-UX gang scheduling, 40, 109

I
-i option, 42, 52
I/O, 166, 181
IMPI, 64
implement
barrier, 14
reduction, 13
improve
bandwidth, 105
coding HP MPI, 111
latency, 105
network performance, 107
improving interhost communication, 57
increase trace magnification, 83
indexed constructor, 15
initialize MPI environment, 4
Initially off field, 98
instrumentation
.mpiview file, 73
.tr file, 79, 91
ASCII profile, 71
counter, 68
creating profile, 68
MPIVIEW, 73–77
multihost, 59
output file, 68
XMPI, 78
instrumentation message bin, 41
interactive mode, 90
intercommunicators, 5
interhost communication
See multiple network interfaces
interoperability problems, 125
interrupt calls to MPI library
See profiling interface
intracommunicators, 5

J
-j option, 31
job ID, 31, 97

K
kill MPI jobs, 61
Kiviat
dialog, 84, 85, 89
views, 89

L
language bindings, 181
language interoperability, 168
latency, 5, 105, 110
-libmtmpi
See linking thread-compliant library
linking thread-compliant library, 30, 170
load sharing facility
See LSF
logical values in Fortran77, 40
LSF (load sharing facility), 64

M
magnify trace log, 83
main window, XMPI, 80, 90
Makefile, 132
man pages
categories, 26
compilation utilities, 26

general HP MPI, 26
HP MPI library, 25
HP MPI utilities, 25
runtime, 26
master_worker.f90, 131
memory leaks, 37
message bandwidth
achieve highest, 110
message bin, 41
message buffering problems, 124
message label, 8
message latency
achieve lowest, 110
message latency/bandwidth, 104, 105
message passing
advantages, 2
message queue, XMPI, 86, 88
message signature analysis, 118
message size, 5
message status, 8
mixed language applications, 168
Monitor interval in second field, 96
MP_GANG, 34, 40
MPI
allgather operation, 10
alltoall operation, 10
app hangs at MPI_Send, 130
broadcast operation, 10
build application on multiple hosts, 21
build application on single host, 20
change execution source, 46
directory structure, 25
gather operation, 10
initialize environment, 4
prefix, 101
routine selection, 110
run application, 19, 31
run application on multiple hosts, 21
run application on single host, 20
scatter operation, 10
terminate environment, 4
MPI 2.0 standard, 166
MPI application, starting, 19
MPI concepts, 3–16
MPI I/O, 166
MPI library extensions
32-bit Fortran, 25
64-bit Fortran, 25
MPI library routines
commonly used, 4
MPI_Comm_rank, 4
MPI_Comm_size, 4
MPI_Finalize, 4
MPI_init, 4
MPI_Recv, 4
MPI_Send, 4
number of, 3
MPI object-space corruption, 118
MPI web sites, xvii
MPI_Abort, 181
MPI_ANY_SOURCE
See improve latency
MPI_Barrier, 13, 14, 111
MPI_Bcast, 4, 11
MPI_Bsend, 7
MPI_Comm
MPI_Comm_c2f, 168
MPI_Comm_rank, 4, 32
MPI_COMM_SELF, 5
MPI_Comm_size, 4
MPI_COMM_WORLD, 5
MPI_COMMD, 34, 35
MPI_Datatype
MPI_Type_f2c, 168
MPI_DEBUG_CONT, 114
MPI_DLIB_FLAGS, 34, 35
MPI_Finalize, 4, 129

MPI_Fint MPI_Comm_c2f, 168
MPI_Fint MPI_Group_c2f, 168
MPI_Fint MPI_Op_c2f, 168
MPI_Fint MPI_Request_c2f, 168
MPI_Fint MPI_Request_f2c, 169
MPI_Fint MPI_Type_c2f, 168
MPI_FLAGS, 34, 37, 104
  using to troubleshoot, 114
MPI_FLAGS options
  DDE, 114
  E, 104
  GDB, 114
  WDB, 114
  XDB, 114
  y, 104
MPI_GET_PROCESSOR_NAME, 181
MPI_GLOBMEMSIZE, 34, 41
MPI_Group MPI_Group_f2c, 168
MPI_handler_function, 181
MPI_Ibsend, 9
MPI_Init, 4, 174
MPI_INSTR, 34, 41, 68
MPI_Irecv, 9
MPI_Irsend, 9
MPI_Isend, 9
MPI_Issend, 9
MPI_LOCALIP, 34, 43
MPI_MT_FLAGS, 44, 45
MPI_NOBACKTRACE, 34
MPI_Op MPI_Op_c2f, 168
MPI_Recv, 4, 8
  high message bandwidth, 110
  low message latency, 110
MPI_Reduce, 12, 13
MPI_Reduce, 13
MPI_REMSH, 45
MPI_ROOT variable, 25
MPI_Rsend, 7
  convert to MPI_Ssend, 40
MPI_Scatter, 12
MPI_Send, 4, 7, 130
  convert to MPI_Ssend, 40
  high message bandwidth, 110
  low message latency, 110
MPI_SHMCNTL, 40
MPI_SHMEMCNTL, 34, 46
MPI_Ssend, 7
MPI_Status_c2f, 169
MPI_Status_f2c, 169
MPI_TMPDIR, 34, 46
MPI_TOPOLOGY
  See also improve network performance
MPI_WORKDIR, 34, 46
MPI_XMPI, 34, 47
mpiCC utility, 28, 29
mpicc utility, 28, 29
mpiclean, 31, 49, 61, 128
mpif77 utility, 28, 29
mpif90 utility, 28, 29
MPIHP_Trace_off, 69, 79
MPIHP_Trace_on, 69, 79
mpijob, 31, 49, 59
mpirun, 49
  appfiles, 55
  command line options, 49–54
  options dialog, 97
  trace file generation, 48
mpirun options fields
  Buffer size, 99
  Initially off, 98

No clobber, 98
Prefix, 98
Simpler trace, 98
Tracing, 97
Verbose, 97
mpirun options trace dialog
Tracing button, 92
mpiview, 49, 62, 73–76
graph analysis functionality, 76
graph types, 73
Window menu, 76
MPMD, 188
MPMD applications, 31, 55
multi_par.f, 131
multilevel parallelism, 16, 110
multiple buffer writes detection, 118
multiple hosts, 21, 55–59
assigning ranks in appfiles, 57
communication, 57
multiple network interfaces, 107
configure in appfile, 107
diagram of, 108
improve performance, 107
using, 107
multiple threads, 16, 110
multi-process debugger, 116

N

Native Language Support (NLS), 65
network interfaces, 107
NLS, 65
NLSPATH, 65
no clobber, 42
See HP MPI abort
No clobber field, 98
nonblocking communication, 6, 9
buffered mode, 9
MPI_Ibsend, 9
MPI_Irecv, 9
MPI_Irsend, 9
MPI_Isend, 9
MPI_Issend, 9
point-to-point, 83
ready mode, 9
receive mode, 9
standard mode, 9
synchronous mode, 9
nonblocking send, 9
noncontiguous and contiguous data, 14
nonportable code, uncovering, 40
number of message copies sent, 88
number of MPI library routines, 3

O

object
convert between languages, 168
one-sided communication, 175
op variable, 13
OPENMP, block partitioning, 149
optimization report, 39
organization of /opt/mpi, 25
over subscribed
See subscription types
overhead process, 83

P

packing and unpacking, 14
parent process, 10
partial trace, 93
peer *See* rank
performance
collective routines, 111
communication hot spots, 57
derived data types, 111
disable argument checking, 40
latency/bandwidth, 104,

105
polling schemes, 111
synchronization, 111
permissions
See rhosts file
ping_pong.c, 131
play
trace file, 84
trace log, 83
PMPI prefix, 101
point-to-point communications
blocking, 83
nonblocking, 83
overview, 5
See also nonblocking communication
See also blocking communication
portability, 3
postmortem mode, 79
prefix
for output file, 68
MPI, 101
PMPI, 101
Prefix field, 92, 98
print HP MPI job ID, 97
problems
`+autodbl`, 29
`+autodbl4`, 29
`+autodblpad`, 29
application hangs at MPI_Send, 130
build, 122
exceeding file descriptor limit, 126
external input and output, 126
Fortran 90 behavior, 125
interoperability, 125
message buffering, 124
performance, 104, 105–111
propagation of environment variables, 124
runtime, 123–126
shared memory, 123
UNIX open file descriptors, 126
process
blocked, 83
colors, 83, 86
hexagons, 90
multi-threaded, 16
overhead, 83
profile in HP MPI, 100
rank, 5, 83
rank of peer process, 87
rank of root, 13
rank of source, 8
reduce communications, 105
running, 83
single-threaded, 16
state, 86, 90
XMPI Focus dialog, 86
process info
view from trace, 85
process placement
multihost, 57
processor subscription, 109
profiling
interface, 101
See also debug HP MPI
See also MPI_FLAGS
using counter instrumentation, 68
using CXperf, 100
using mpiview, 73, 77
using XMPI, 78
progression, 106
propagation of environment variables, 124
pthreads, 30, 170

R

race condition, 114
rank, 5
of calling process, 4
of root process, 13
of source process, 8
reordering, 39

raw trace files, 129
ready send mode, 7
rebuild Xresource database, 163
receive
  message information, 8
  message methods, 6
  messages, 4, 5
receive buffer
  address, 13
  data type of, 13
  data type of elements, 8
  number of elements in, 8
  starting address, 8
recvbuf variable, 12, 13
recvcount variable, 12
recvtype variable, 12
red
  See process colors
reduce, 12
reduce-scatter, 12
reduction, 13
  operation, 13
release notes, 25
remote hosts
  See rhosts file
remote shell, 21
  `remsh` command, 45, 122
  secure, 45
remsh, 21
`remsh` command
  See remote shell
reordering, rank, 39
req variable, 9
rewind trace log, 83
rhosts file, 21, 122
ROMIO
  See I/O, 166
root process, 10
root variable, 11, 12, 13
rotate graph, 76
routine selection, 110
run
  appfile interactively, 90
  application, 19
  MPI application, 31, 123
  MPI on multiple hosts, 21, 49, 55–59
  MPI on single host, 20
  MPI on single hosts, 49
  mpiview, 73
  process, 83
  XMPI, 61
  See also mpirun
run examples, 132
runtime
  environment variables, 34
  problems, 123–126
  utilities, 26, 49–62
  utility commands, 49
  mpiclean, 61
  mpijob, 59
  mpirun, 49
  mpiview, 62
  xmpi, 61
runtime environment variables
  MP_GANG, 34, 40
  MPI_COMMD, 34, 35
  MPI_DLIB_FLAGS, 34, 35
  MPI_FLAGS, 34, 37
  MPI_GLOBMEMSIZE, 34, 41
  MPI_INSTR, 34, 41
  MPI_LOCALIP, 34, 43
  MPI_MT_FLAGS, 44, 45
  MPI_NOBACKTRACE, 34
  MPI_REMSH, 45
  MPI_SHMCNTL, 40
  MPI_SHMEMCNTL, 34, 46
  MPI_TMPDIR, 34, 46
  MPI_WORKDIR, 34, 46
  MPI_XMPI, 34, 47

S

s, 38

scan, 12
scatter, 10, 12
secure shell, 45
select process, 86
select reduction operation, 13
send buffer
address, 13
data type of, 13
number of elements in, 13
sendbuf variable, 12, 13
sendcount variable, 12
sending
data in one operation, 4
messages, 4–6
process rank, 88
sendtype variable, 12
setenv
MPI_ROOT, 25
XAPPLRESDIR, 163
setting up view options, 95
shared libraries, 54
shared memory
control subdivision of, 46
default settings, 40
MPI_SHMEMCNTL, 46
specify, 41
system limits, 123
SIGBUS, 119
SIGILL, 119
SIGSEGV, 119
SIGSYS, 119
Simpler trace field, 98
single-process debuggers, 114
single-threaded processes, 16
SMP, 189
snapshot utility, 91
source variable, 8, 9
spin/yield logic, 39
SPMD, 190
SPMD applications, 31
src
See sending process rank
src field, 88
standard send mode, 7
starting
HP MPI, 19, 122
multihost applications, 21, 122
singlehost applications, 20
status, 8
status variable, 8
stdargs, 181
stdin, 126
stdio, 126, 181
stdout, 126
stop playing trace log, 83
storing temp files, 46
structure constructor, 15
subdivision of shared memory, 46
subscription
definition of, 109
types, 109
swapping overhead, 41
synchronization, 13
performance, and, 111
variables, 3
synchronous send mode, 7

T

-t option, 48, 53
tag
See tag argument value
tag argument value, 87, 88
tag field, 87, 88
tag variable, 8, 9
terminate MPI environment, 4
thread
communication, 62
multiple, 16
safety, 170
thread-compliant library, 30, 170
+Oparallel, 30

total transfer time, 5
TOTALVIEW, 48
TotalView
See multi-process debugger
trace
get full, 93
get partial, 93
view process info, 85
Trace dialog, 92
trace file
create, 79
Kiviat, 89
play, 84
state, 84, 85, 89
viewing, 80
trace file generation
enable runtime, 91
enable runtime raw, 92, 97
using
mpirun, 48
XMPI, 48
trace log
fast forward, 83
magnification, 83
play, 83
rewind, 83
set magnification, 83
stop playing, 83
trace magnification, 83
Trace Selection dialog, 81
tracing
See trace file generation
Tracing button, 92
Tracing field, 97
tracing options dialog
See mpirun options fields
troubleshooting, 113
Fortran 90, 125
HP MPI, 121–130
message buffering, 124
MPI_Finalize, 128
mpiclean, 31
mpijob, 31
UNIX file descriptors, 126
using MPI_FLAGS, 114
using the what command, 18, 121
version information, 18, 121
See MPIHP_Trace_off
See MPIHP_Trace_on
tuning, 103–111
twisted-data layout, 149

U

under subscribed
See subscription types
UNIX open file descriptors, 126
unpacking and packing, 14
using
counter instrumentation, 68
gang scheduling, 40
mpiview, 62, 73–77
multiple network interfaces, 107
profiling interface, 101
XMPI in interactive mode, 90, 95
XMPI in postmortem mode, 79, 80

V

variables
buf, 7, 8, 9, 11
comm, 8, 9, 11, 12, 13
count, 7, 8, 9, 11
dest, 8, 9
dtype, 8, 9, 11, 13
MPI_DEBUG_CONT, 114
MPI_ROOT, 25
op, 13
recvbuf, 12, 13
recvcount, 12
recvtype, 12
req, 9
root, 11, 12, 13

runtime, 34–40
sendbuf, 12, 13
sendcount, 12
sendtype, 12
source, 8, 9
status, 8
tag, 8, 9
XAPPLRESDIR, 163
vector constructor, 15
Verbose field, 97
verbose mode, 97
version, using what, 18, 121
View, 82
view
kiviat information, 89
multiple mpiview graphs, 76
process info, 85
trace file, 80
view options
changing and setting, 95
viewing
ASCII profile, 69
instrumentation file, 73–77
trace file, 80–99

W

WDB, 37, 114, 130
what command, 18, 121

X

X resource environment variable, 163
XAPPLRESDIR, 163
XDB, 37, 114, 130
XMPI, 78–99
Application Browser dialog, 90
buffer size dialog, 96
command line syntax, 61
Confirmation dialog, 93
Datatype dialog, 87
display, 78
Express dialog, 93
Focus dialog, 86
Focus dialog message queue, 88
Focus dialog select process, 86
interactive mode, 78
Kiviat dialog, 89
main window, 80, 90
monitor options dialog, 95
postmortem mode, 78, 79
rebuild Xresource database, 163
resource file, 163
snapshot utility, 91
trace
application default settings, 25
Trace dialog, 92
Trace dialog View, 82
trace file generation, 48
Trace Selection dialog, 81
using interactively, 90, 95
X application resource environment variable, 163
X resource file contents, 164
XAPPLRESDIR, 163
xmpi command line, 49, 61
XMPI Focus fields
cnt, 87, 88
comm, 87, 88
copy, 88
peer, 87
src, 88
tag, 87, 88
XMPI monitor options field
Automatic snapshot, 95
Monitor interval in second, 96
XMPI Trace
dialog, 82, 90
Dump, 92

Express, 92

Y

yellow
See process colors
yield/spin logic, 39

Z

zero-buffering, 40